

3

Transforming Files

Let's start our "deep dive" into the world of Talend Open Studio for data integration by looking at some common file integration techniques. Of all the methods of integrating systems, using files and file exchanges is probably one of the oldest and most common. With the rise of other, more modern, integration methods over the last few years, web services for example, integrating files might be seen as a bit unfashionable. But don't be fooled; exchanging files between systems can be an excellent way to integrate. To support this, lots and lots of applications have file-based integration APIs.

In this chapter, we will learn how to:

- Transform files from one format to another
- Use the Studio's expression editor to modify data
- Build advanced and multi-schema XML files
- Use lookups to enrich data
- Get familiar with the Studio development environment by following the detailed step-by-step examples

Transforming XML to CSV

Let's start with a simple file format transformation. Many modern applications use **Extensible Markup Language (XML)** formats to get data in and out. Other, often simpler, systems use a **Comma Separated Format (CSV)**. Common, desktop-based systems, such as Excel and Access, have wizards for taking data in the CSV format. We'll work through the process of taking a simple XML file and extracting its data into a comma-separated format.

Before we dive in and actually start to configure a the Studio job, let's look at the data that we want to transform. Our input file is an XML product catalogue named `catalogue.xml`, which is present in the datafiles of this chapter. Open this in the XML viewer of your choice. You can see that the data is pretty self-explanatory. The file contains data about **Stock Keeping Units (SKUs)**. There are a number of repeating SKU elements, each containing an `skuid`, `skuname`, `size`, `colour`, and `price`.



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <catalogue>
3   <sku>
4     <skuid>1233212406</skuid>
5     <skuname>Summer Dress</skuname>
6     <size>6</size>
7     <colour>Green</colour>
8     <price>39.99</price>
9   </sku>
10  <sku>
11    <skuid>1233212408</skuid>
12    <skuname>Summer Dress</skuname>
13    <size>8</size>
14    <colour>Green</colour>
15    <price>39.99</price>
16  </sku>
17  <sku>
18    <skuid>1233212410</skuid>
19    <skuname>Summer Dress</skuname>
20    <size>10</size>
21    <colour>Green</colour>
22    <price>39.99</price>
23  </sku>
```

We want to extract this data into a spreadsheet-style format with similar columns to the XML elements, that is `skuid`, `skuname`, `size`, `colour`, and `price`.

So let's create the job step-by-step:

1. Create a new job in the Studio and name it `XML2CSV`. (I've also created a new folder named `Chapter3` to contain the jobs for this chapter and I'll follow this convention for other chapters throughout the book.)

- Next, we'll define the metadata for the `catalogue` file we want to read. In the **Repository** window, expand the **Metadata** section, right-click on **File XML**, and select **Create file xml**. The **New Xml File** wizard will open:

New Xml File

File - Step 1 of 5

Name is empty.

Name

Purpose

Description

Author

Locker

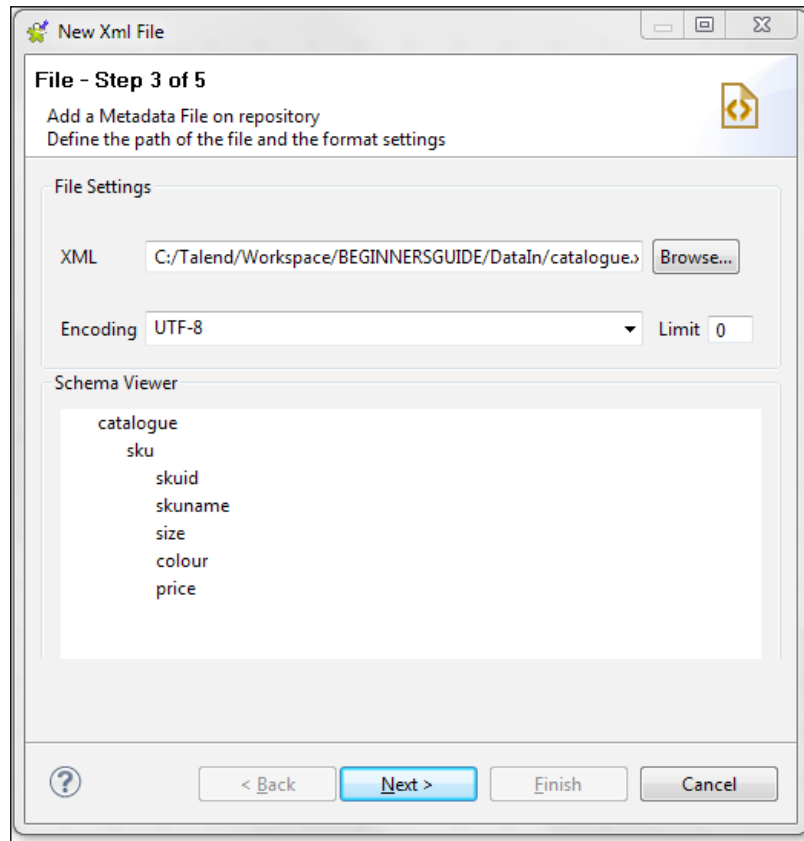
Version

Status

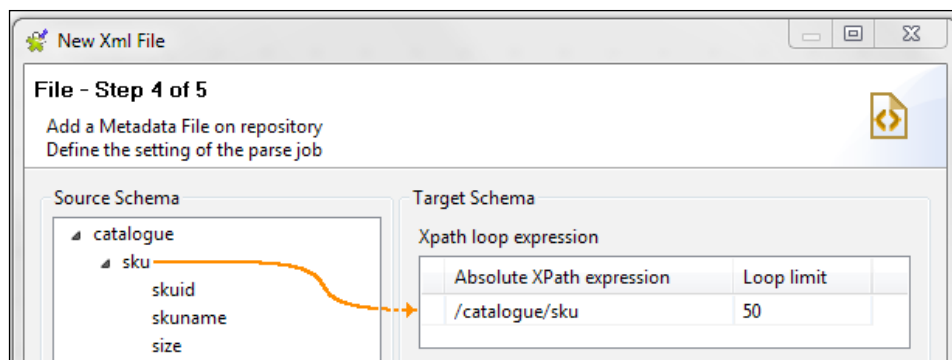
Path

- Enter `catalogue` into the **Name** field and click on **Next**.
- As we are creating metadata for a data source or input to the process, select **Input XML** on the step 2 screen and click on **Next**.

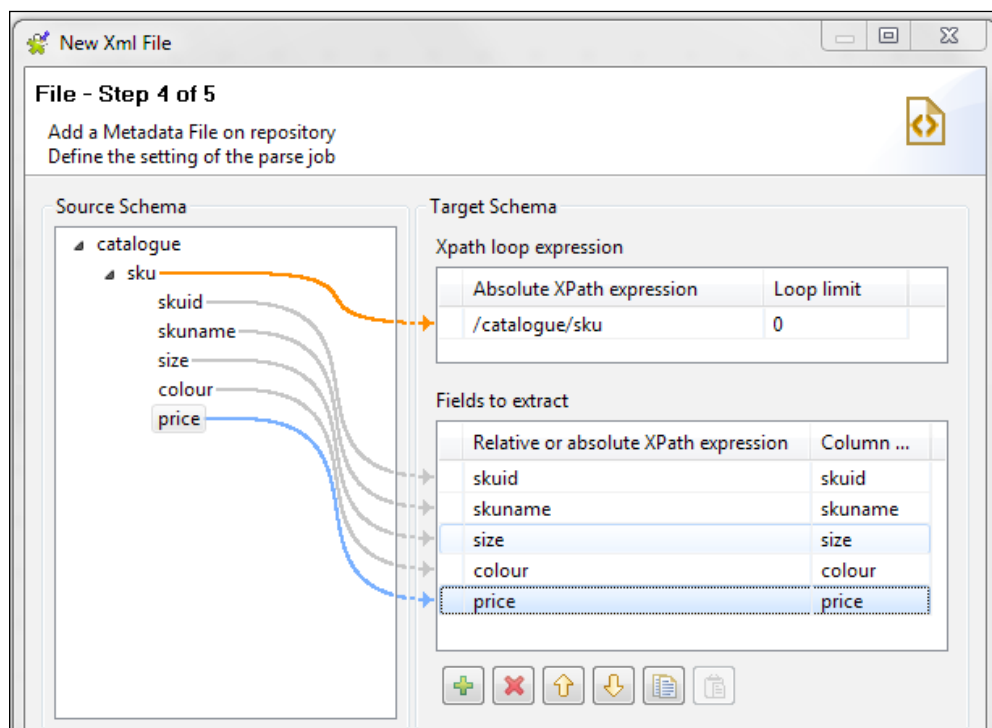
5. In step 3, we need to provide an XSD file or a sample XML file so that the Studio can determine the XML schema. In our case, we will provide a sample XML file. Click on **Browse** and navigate to the `catalogue.xml` file. Select it and click on **Open**. The Studio will determine the file encoding automatically and will provide a view of the schema structure. Click on **Next** to move to step 4:



6. Step 4 of the process is where we determine how the Studio should read the XML files. Specifically, the Studio needs to know which elements to loop on and which elements should be extracted. Let's deal with the loop first.
7. Our catalogue file is a series of SKUs. So, in order to extract data for each SKU, we need to set this as our loop element. We want the Studio to loop over all of the SKUs when the job runs. In order to configure this, we need to map the SKU to the **Xpath loop expression** box. Click on the SKU element in the **Source Schema** pane and drag it to the **Xpath loop expression** box:



8. The **Loop limit** field determines how many times the job will loop over the selected element. By default, this is configured to **50**, but let's change this to **0**, which is the number used to configure no limit.
9. Now, we can configure the fields we want to extract from the catalogue file. Drag the **skuid**, **skuname**, **size**, **colour**, and **price** fields to the **Fields to extract** pane:





XPath is a query language for finding information in an XML document. It uses path expressions to navigate to the required elements. XPath is a big subject in its own right, and it is outside the scope of this book to go into detail on the subject. However, W3C Schools provides an excellent tutorial on the subject at <http://www.w3schools.com/xpath/>, and readers may wish to take some time to read through this.

10. Click on the **Refresh Preview** button in the bottom left-hand corner of the window, and the Studio will read the file and preview the extract of the data in a tabular format. Click on **Next** to move to step 5.

skuid	skuname	size	colour	price
1233212406	Summer Dress	6	Green	39.99
1233212408	Summer Dress	8	Green	39.99
1233212410	Summer Dress	10	Green	39.99
1233212412	Summer Dress	12	Green	39.99

11. Step 5 of the metadata setup allows us to confirm or alter the schema as determined by the wizard and add this to the **Repository** window. The Studio has determined data types, field length, and precision values. In this case, we can make a couple of changes. Our **skuid** field is a unique key, so we can note as such in the schema by checking the **Key** checkbox for the **skuid** row. As **skuid** is a key, we would need it to be present for every record, so it cannot be nullable. Uncheck the **Nullable** checkbox for the **skuid** row. Finally, for this step, change the **Name** field to catalogue. Click on **Finish**:

New Xml File

File - Step 5 of 5

Add a Schema on repository
Define the Schema


Name

Comment

Schema

Click to update schema preview

Description of the Schema

Column	Key	Type	<input checked="" type="checkbox"/> N..	Date Pattern (Ctrl...	Length	Precision	Default	Comment
 skuid	<input checked="" type="checkbox"/>	long	<input type="checkbox"/>		10	0		
skuname	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>		12	0		
size	<input type="checkbox"/>	Integer	<input checked="" type="checkbox"/>		2	0		
colour	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>		5	0		
price	<input type="checkbox"/>	Float	<input checked="" type="checkbox"/>		5	3		




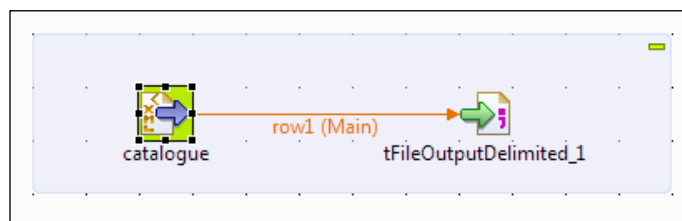
Often, XML files will come with an **XML Schema Definition (XSD)** file so that data types can be determined directly. If you don't have an XSD file, then some assumptions can be made from the data we have. Although, it is important to bear in mind that sample data sets may not give the full picture, so some caution is advised here. For example, the metadata wizard has determined that size is an integer based upon the data presented in the file, but we might subsequently come across the sizes of *S*, *M*, and *L*, making the data type incorrect. If you are using an XML file as the basis for the schema, ensure that it covers all of the data types you would expect or, alternatively, edit the schema definition as shown previously in step 5.

12. With our metadata configured, we can now build the job. From the **Repository** window, expand the **Metadata** section, click on the newly created catalogue metadata and drag it onto the Job Designer. You'll be presented with a list of components compatible with the metadata we have defined. In this case, we can observe two components, namely **tFileInputXML** and **tExtractXMLField**. Our job needs the **tFileInputXML** component, so select this and click on **OK**. The component will now be visible on the Job Designer.
13. In the **Component** palette, search for `delimited` and find the **tFileOutputDelimited** component (it is in folder **File | Output**). Drag this onto the Job Designer next to the **tFileXMLInput** component.



14. We then need to connect the two components. There are a number of different connection types available in the Studio. For this job, we will use the **Main** connector. This connection type is probably the most used in the Studio and represents a straightforward passing of data between two components. To connect it in this way, right-click on the **tFileInputXML** component and select **Row | Main**. This will create a line from the component to your cursor. Place your cursor over the **tFileOutputDelimited** component and click on it. This will create a connection line between the two components. (It will also merge the gray background box, signifying that the two components are now part of a larger integration process.)

 Note that the process of connecting the two components will synchronize the schemas from the catalogue XML component to the delimited output component.

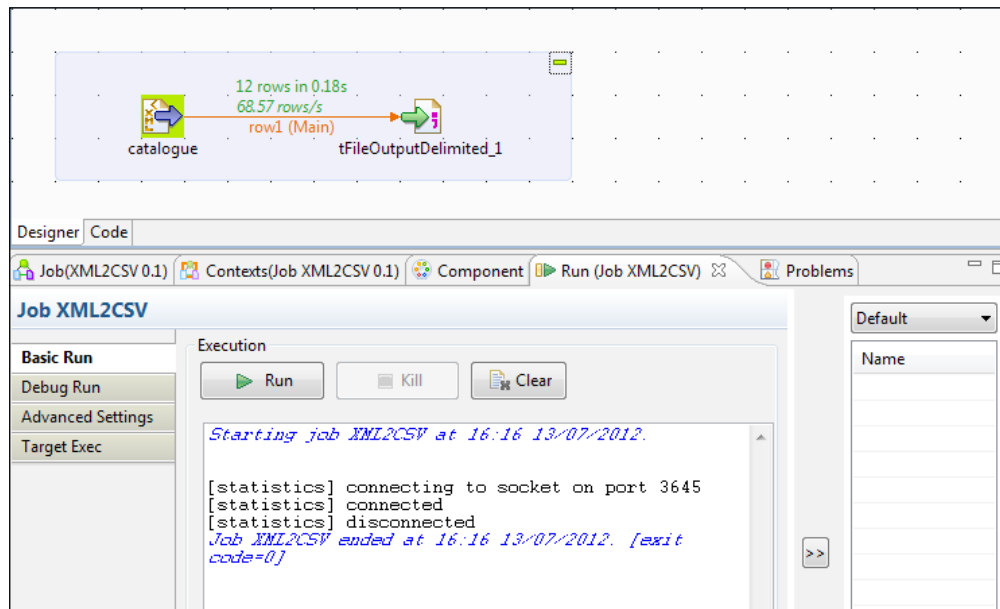


15. Let's now configure the **tFileOutputDelimited** component. Click on the component, and select the **Component** tab in the panel below the Job Designer. Some default configurations are presented, and we'll change a few of these.
16. Change the filename to the full path of the output file you require. This might be something similar to `C:/Talend/Workspace/GETTINGSTARTED/DataOut/Chapter3/catalogue-out.csv`.

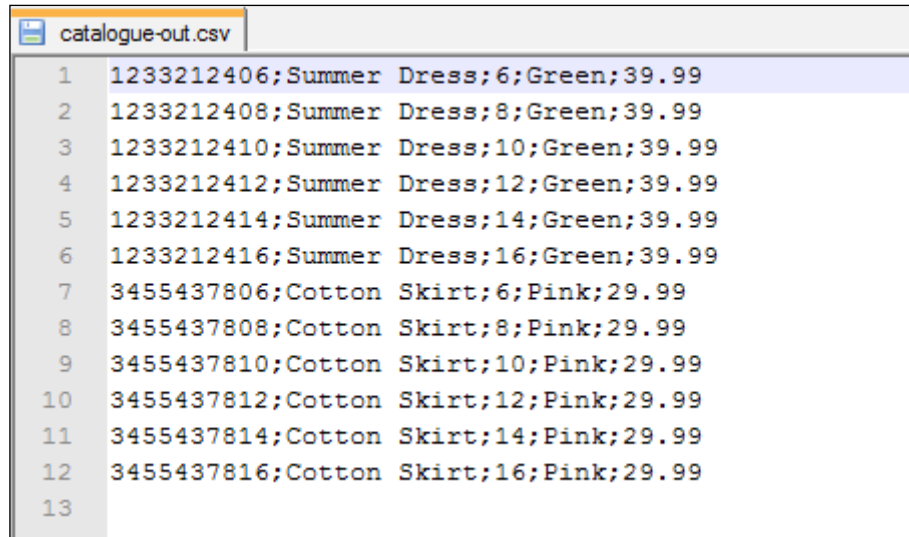
We're now ready to test our job!

In the bottom panel below the Job Designer, click on the **Run** tab and then click on the **Run** button. This will save the configurations you have made, compile the job, and then run it.

The **Run** tab will show some statistics as the job executes and, if a runtime error occurs, will show error details to allow you to diagnose any issues. The Job Designer will also indicate how many rows of data have been extracted for the input file we are working with.



Browse to your output folder, and you will see the output file similar to the one shown in the following screenshot:

A screenshot of a text editor window titled 'catalogue-out.csv'. The window displays a list of 12 items, each on a new line. The items are numbered 1 through 12 on the left margin. Each item consists of a product ID, a category, a name, a size, a color, and a price, separated by semicolons. The first six items are 'Summer Dress' items with IDs 1233212406 through 1233212416, all priced at 39.99. The next six items are 'Cotton Skirt' items with IDs 3455437806 through 3455437816, all priced at 29.99. The 13th line is empty.

1	1233212406;Summer Dress;6;Green;39.99
2	1233212408;Summer Dress;8;Green;39.99
3	1233212410;Summer Dress;10;Green;39.99
4	1233212412;Summer Dress;12;Green;39.99
5	1233212414;Summer Dress;14;Green;39.99
6	1233212416;Summer Dress;16;Green;39.99
7	3455437806;Cotton Skirt;6;Pink;29.99
8	3455437808;Cotton Skirt;8;Pink;29.99
9	3455437810;Cotton Skirt;10;Pink;29.99
10	3455437812;Cotton Skirt;12;Pink;29.99
11	3455437814;Cotton Skirt;14;Pink;29.99
12	3455437816;Cotton Skirt;16;Pink;29.99
13	

Success! Let's move on to another file transformation example, from CSV to XML.

Transforming CSV to XML

Working the other way around, we will now transform a simple CSV file into an XML format. Again, this is a common integration scenario, where, for example, data needs to move from a spreadsheet to a format suitable for loading into another application via its native API.

Start by creating a new job in the `Chapter3` folder and call it `CSV2XML`. We're going to use the CSV output from our previous task as an input file this time, so take a copy of `catalogue-out.csv` from your `DataOut` folder and drop it into the `DataIn` folder. To avoid confusion, let's rename this to `csv2xml-catalogue.csv`.

As in the previous example, we'll start by declaring the metadata for the input file before completing the main configuration of the job. Perform the following steps:

1. In the **Metadata** section of the **Repository** window, right-click on **File delimited** and select **Create file delimited**. The metadata wizard window will appear. Enter `csv_catalogue` into the **Name** field and click on **Next**:

New Delimited File

File - Step 1 of 4

Add a Metadata File on repository
Define the properties

Name:

Purpose:

Description:

Author:

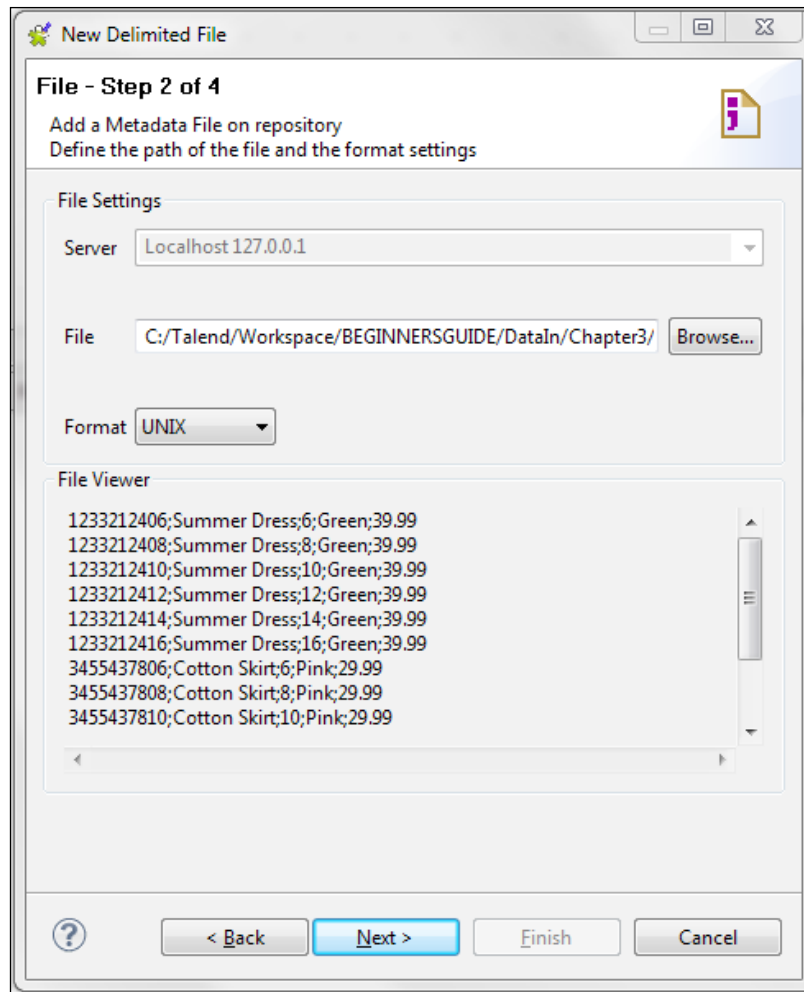
Locker:

Version: ☐ ☐

Status:

Path:

2. In step 2, we need to add our sample input file so that the schema can be determined. Click on the **Browse** button and navigate to the sample file in our DataIn folder. The wizard will show a preview of the file in the **File Viewer** pane:



3. Click on **Next** to view the parse settings for the file. Here we can define field and row separators, and header and footer rows, for example. The wizard will have made a guess at the settings based on the file presented, and we'll keep the selections it has made. Click on **Next** to move to step 4.

- In the final step, we can set the schema definition. Again, the wizard will have made a best guess based on the file presented, but there are a few things that we need to change. The columns do not have meaningful names, so modify them to **skuid**, **skuname**, **size**, **colour**, and **price**. We'll also make **skuid** a key field and not nullable. Finally, change the **Name** field to be **csv_catalogue**. When finished, your settings should be as follows:

File - Step 4 of 4
Add a Schema on repository
Define the Schema

Name:
Comment:

Schema
Click to update schema preview

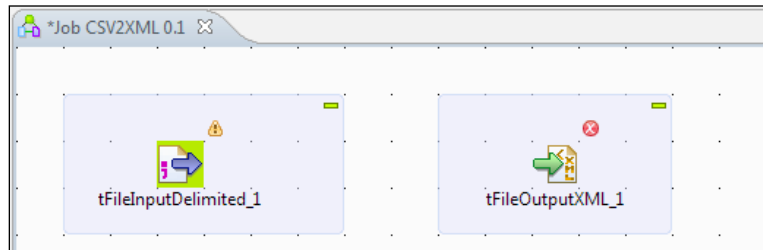
Description of the Schema

Column	Key	Type	Nullable	Date Pattern (Ctrl...)	Length	Precision	Default	Comment
skuid	<input checked="" type="checkbox"/>	long	<input type="checkbox"/>		10	0		
skuname	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>		12	0		
size	<input type="checkbox"/>	Integer	<input checked="" type="checkbox"/>		2	0		
colour	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>		5	0		
price	<input type="checkbox"/>	Float	<input checked="" type="checkbox"/>		5	3		

< Back Next > Finish Cancel

- Click on **Finish** to complete the metadata configuration.
- Now click on the file delimited metadata we have created and drag it onto the Job Designer. As before, we are presented with a list of components that can be used with the metadata we have defined. We want a simple delimited file input, so select the **tFileInputDelimited** component and click on **OK**.

- Let's now configure the output side of this job. Search for `xml` in the palette, and locate the **tFileOutputXML** component. Drag this onto the Job Designer next to the delimited file input:




- We can now join the two components together as before. Right-click on the delimited file input and select **Row | Main**. Drop the line attached to your cursor onto the XML output component by clicking on it.

We need to add a few more configurations to the XML output component.

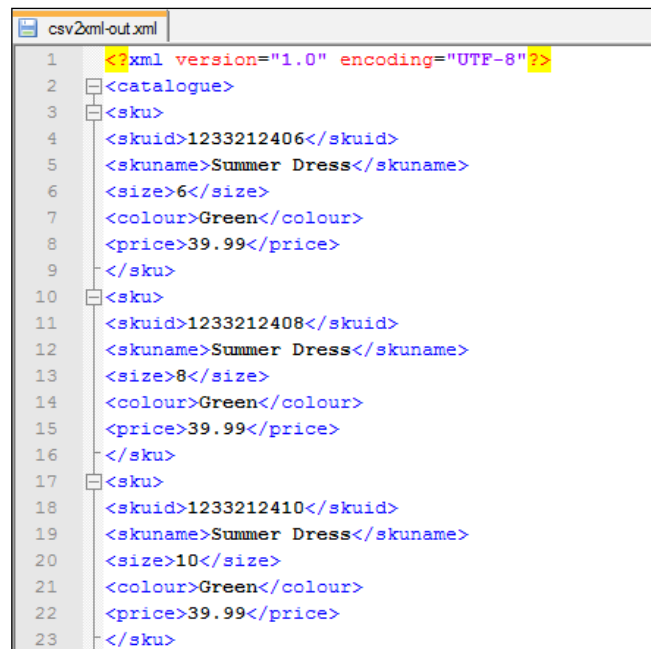
- Under **Basic settings**, enter the output filename and path we require (or browse to it by clicking on the ellipsis button).
- The next configuration item, **Row tag**, is the name of the XML element that will wrap around each row of data. If you are working to a predefined XML schema, then you can determine this from the XSD. In this case, let's set this to `sku`, as each row of our data is an SKU and its associated data.
- Moving onto the **Advanced settings** tab, we'll need to add a root tag to the XML output. Again, this might be determined from the XML schema, but in our case, let's use the value `catalogue`. Click on the **+** button of the **Root tags** pane to add a new row and then change the **newline** value to `catalogue`.
- Let's also change the file encoding to **UTF-8**.

Again, we have accepted the default values for a number of the configuration settings and we'll revisit some of these in different scenarios throughout the book.

Let's run the job. Go to the **Run** tab in the bottom panel, and click on the **Run** button to save, compile, and run the job.

 You can also run jobs by pressing the *F6* button on your keyboard. It's a useful shortcut when you are running jobs a number of times during the development/debug phase.

Go to your DataOut folder, and you should see the XML output file as shown in the next screenshot:



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <catalogue>
3   <sku>
4     <skuid>1233212406</skuid>
5     <skuname>Summer Dress</skuname>
6     <size>6</size>
7     <colour>Green</colour>
8     <price>39.99</price>
9   </sku>
10  <sku>
11    <skuid>1233212408</skuid>
12    <skuname>Summer Dress</skuname>
13    <size>8</size>
14    <colour>Green</colour>
15    <price>39.99</price>
16  </sku>
17  <sku>
18    <skuid>1233212410</skuid>
19    <skuname>Summer Dress</skuname>
20    <size>10</size>
21    <colour>Green</colour>
22    <price>39.99</price>
23  </sku>
```

So, our first two jobs have taken us on a quick round-trip from XML to CSV and back again. They have also illustrated that the Studio can perform integration transformations (admittedly, quite simple transformations thus far) by configuring, rather than coding. Let's make things a little more complex by introducing the Studio's tMap component, which among other things, allows us to perform functions against fields, using its expression editor.

Maps and expressions

In most integration scenarios, we are unlikely to find that all data fields can be passed from one system to another without any modification. Because different systems model the same objects in different ways, there's often the need, to not only change the file format, but also to change the data model and content in some way.

For our next job design, we'll do another CSV to XML transformation; but this time, the data models of the input and the output (and hence the schemas) will be different. We'll use the Studio's mapping component and Expression editor to help us deal with these differences.

To start off, let's look at our two data models to examine the differences. Our CSV file is a customer datafile and has the following fields:

- Customer ID
- First Name
- Last Name
- Address1
- Address2
- Town City
- County
- Postcode
- Telephone

We know that all of these fields have a string data type and that all fields are mandatory, except for Address2.

The XML file we want to produce has a similar, yet different set of fields, as follows:

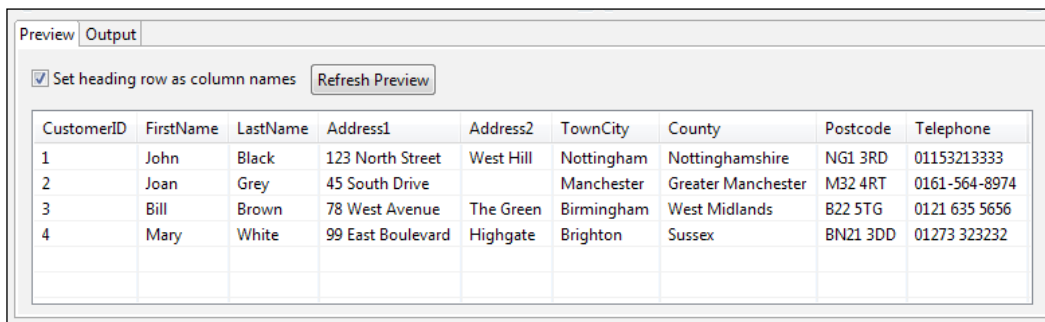
- id (left-padded with zeros to make its length 8)
- name (for example, John Smith)
- address_1 (for example, house number, street name, and district)
- address_2 (any other address fields)
- telephone_number (must be numbers only, without spaces or other characters)

Our first decision, then, is completely outside of the Studio. How are we going to map the CSV fields to the XML fields? The following table shows the mappings we will make for this integration job; although, of course, other mappings could be equally valid:

XML Field	CSV Mapping
id	Customer ID field (left-padded with zeros to make its length 8)
name	First Name and Last Name
address_1	Address1 and Address2
address_2	Town City, County, and Postcode
telephone_number	Telephone with non-numeric characters removed

Rather than trying to get everything right on the first go, let's approach this in an iterative way, implementing the transformations we need, step by step, and checking the results as we go:

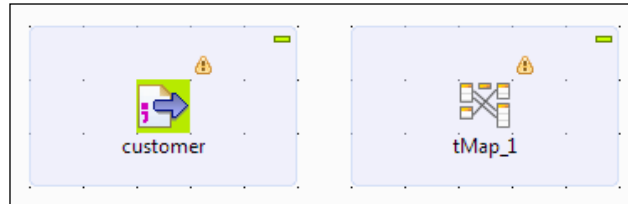
1. Let's create a new job and call it `Expressions` in the `Chapter3` folder of the Studio. There is a sample datafile for our CSV schema (`expressions.csv`). Copy that into your `DataIn` folder.
2. As in our previous examples, we'll start by creating the metadata for the input file. Right-click on **File delimited** in the **Metadata** section of the **Repository** window and select **Create file delimited**. As our input file contains customer data, let's name this metadata definition `customer`. Follow the same steps that we took in the previous example job to define the metadata, using the `expressions.csv` file.
3. As we have seen, the Studio does a good job of guessing the settings, but in step 3 of the configuration, we can make some changes. Our sample file contained column names in the first row, so in the **Preview** tab of step 3, we can configure the metadata to reflect this. Check the checkbox named **Set heading row as column names** and then click on the **Refresh Preview** button. Our column headers are now populated by the first row of data:



CustomerID	FirstName	LastName	Address1	Address2	TownCity	County	Postcode	Telephone
1	John	Black	123 North Street	West Hill	Nottingham	Nottinghamshire	NG1 3RD	01153213333
2	Joan	Grey	45 South Drive		Manchester	Greater Manchester	M32 4RT	0161-564-8974
3	Bill	Brown	78 West Avenue	The Green	Birmingham	West Midlands	B22 5TG	0121 635 5656
4	Mary	White	99 East Boulevard	Highgate	Brighton	Sussex	BN21 3DD	01273 323232

4. In step 4 of the wizard, you can see that the column names have been carried forward as the schema column names. As previously, let's make the **id** field, **CustomerID**, a key and not nullable. Click on **Finish** to complete the metadata configuration.
5. Now click on our newly created delimited metadata and drag it onto the Job Designer. Choose **tFileInputDelimited** from the component list.

6. Search for `tMap` in the **Palette** window and drop this onto the Job Designer. The **tMap** component is used to map data inflows to data outflows, and we'll use this to define the mapping we have noted previously:

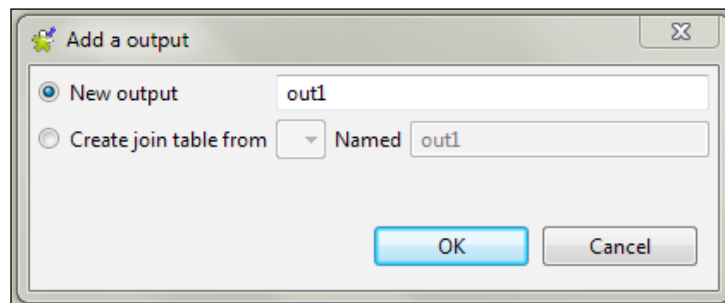


7. Right-click on the delimited customer input, and select **Row | Main** and join it to the **tMap** component.
8. Double-click on the **tMap** component and the **Map Editor** will appear. The **Map Editor** is a powerful tool that allows you to map, transform, and route the dataflows within your job.




The **Map Editor** has a number of panels for configuring mappings and transformations. The input panel on the top left-hand side shows all of the incoming dataflows. The variable panel in the middle allows you to configure mappings to variables. The output panel on the top right-hand side is used to define the outgoing dataflows and the mappings to them. At the bottom of the **Map Editor** is the **Schema editor**, which shows a schema view of all the inputs and outputs. Behind the **Schema editor** is the **Expression editor**, which allows you to edit any of the expressions used.

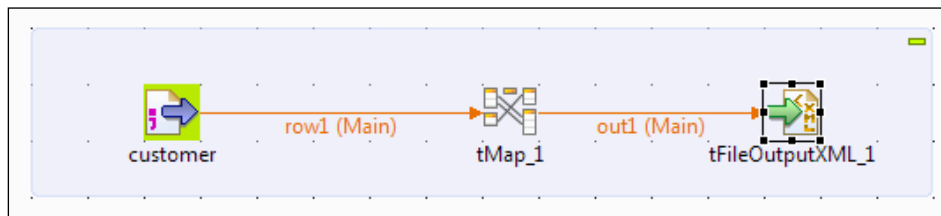
9. Our first step in the **Map Editor** is to configure the data output that we need. In the output pane on the right-hand side, click on the **+** button to create a new output. You'll be presented with a pop-up window to name the output:



10. Let's accept the default **New output** name by clicking on **OK**. A new output is created in the output pane and a new, empty schema is created in the schema pane.
11. Click on the **+** button of the **out1** schema five times, to add new rows to the schema. Change the names of the new rows added to match our output schema, as defined in the previous table. The **id** field needs to be of data type integer, to match the input data coming from the delimited file. Let's also make the **id** field of the new output a key and not nullable:

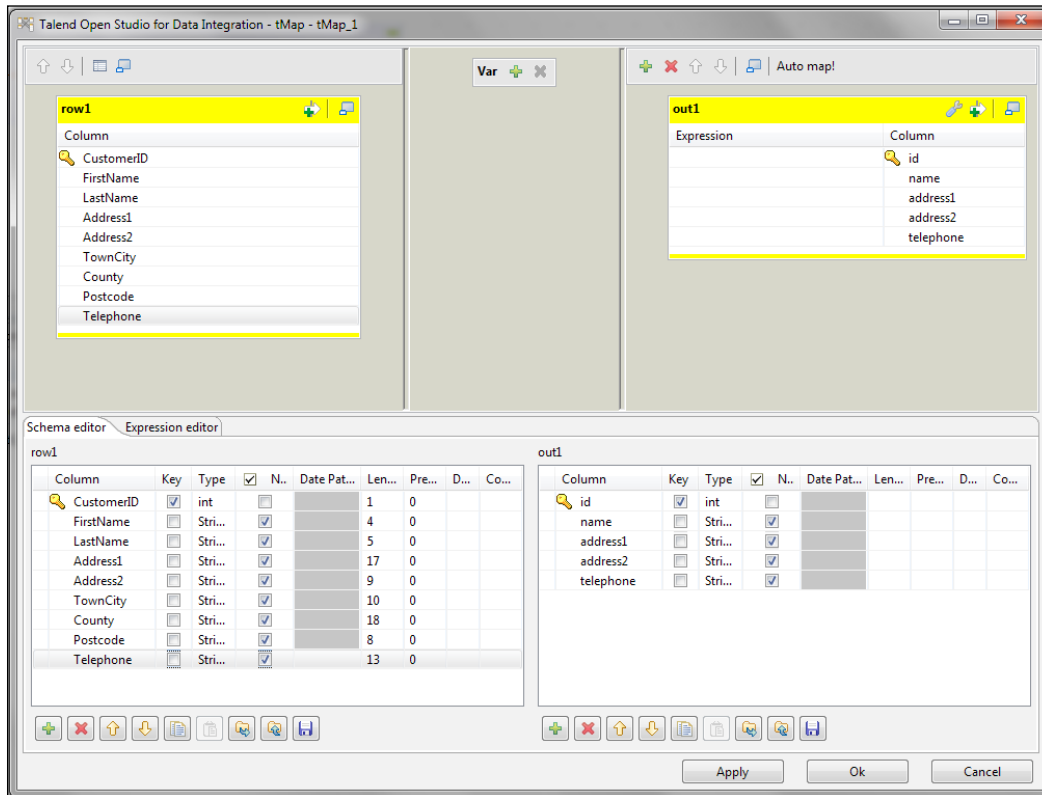
out1				
Column	Key	Type	<input checked="" type="checkbox"/>	Nullable
 id	<input checked="" type="checkbox"/>	int	<input type="checkbox"/>	<input type="checkbox"/>
name	<input type="checkbox"/>	String	<input type="checkbox"/>	<input checked="" type="checkbox"/>
address1	<input type="checkbox"/>	String	<input type="checkbox"/>	<input checked="" type="checkbox"/>
address2	<input type="checkbox"/>	String	<input type="checkbox"/>	<input checked="" type="checkbox"/>
telephone	<input type="checkbox"/>	String	<input type="checkbox"/>	<input checked="" type="checkbox"/>

12. Before we configure the mappings in the **Map Editor**, let's configure the XML output. In the **Palette** window, search for `xml` and drag a **tFileOutputXML** component onto the Job Designer.
13. Right-click on the **tMap** component, select **Row | out1**, and drop the connector onto the XML output component:

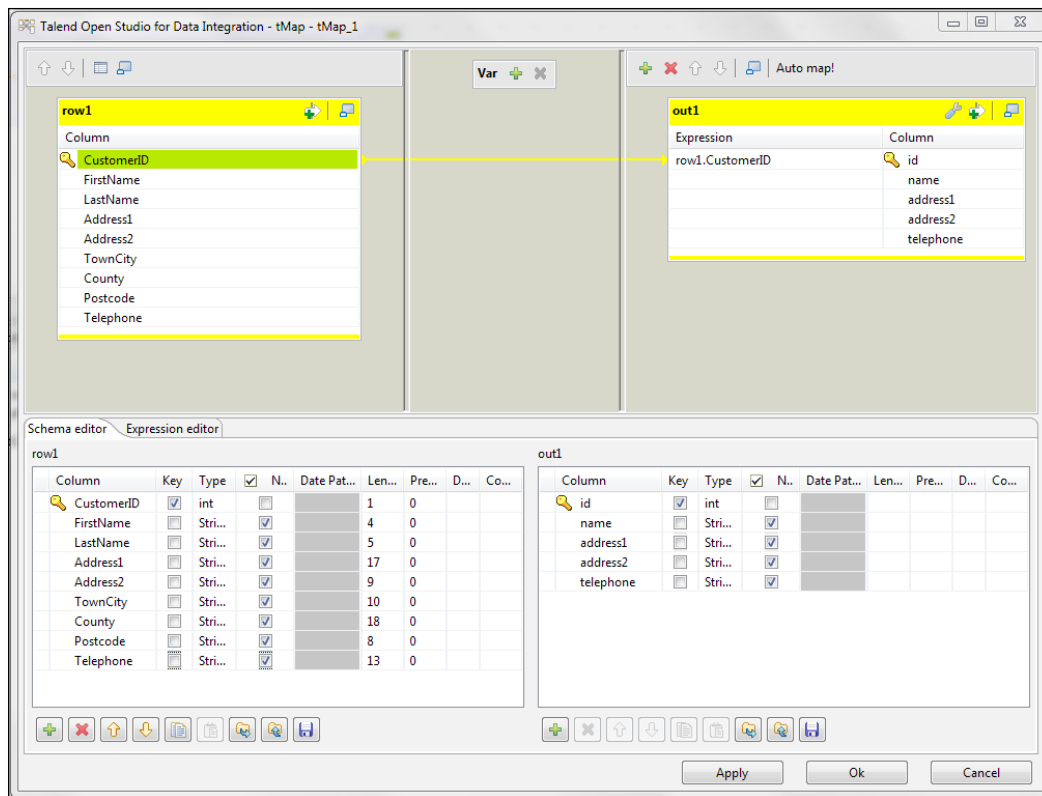


14. Click on the XML component and then the **Component** tab to reveal its configuration settings. We will make some changes to the default settings. Starting with **Basic settings**, set the **File Name** field to something appropriate in your DataOut folder.
15. Change the **Row tag** field to `customer` (as each row will be a customer).
16. Moving to the **Advanced settings** tab, add a **Root tag** field and name it `customers`.

17. Change the **Encoding** field to **UTF-8**. So far, so good. We now need to define the mappings in the **Map Editor**. Double-click on the **tMap** component to open this.



The mapping functionality is supported by a simple, drag-and-drop interface. Let's try this out by clicking on the **CustomerID** field under **row1**, and dragging it to the **Expression** column on the **id** row of the **out1** schema:



The component draws a line between the two fields, as shown in the previous screenshot, and also fills in the **Expression** column of the **out1** schema with **row1.CustomerID**. You could also type this directly into the **Expression** column rather than using the drag-and-drop option, if you prefer.

Before we go any further, let's click on **OK** on the mapping window and run the job to check that everything is connected. The job will save, compile, and run, and you should see the following output in your DataOut folder:



```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <customers>
3  <customer>
4    <id>1</id>
5    <name></name>
6    <address_1></address_1>
7    <address_2></address_2>
8    <telephone_number></telephone_number>
9  </customer>
10 <customer>
11   <id>2</id>
12   <name></name>
13   <address_1></address_1>
14   <address_2></address_2>
15   <telephone_number></telephone_number>
16 </customer>
17 <customer>
18   <id>3</id>
19   <name></name>
20   <address_1></address_1>
21   <address_2></address_2>
22   <telephone_number></telephone_number>
23 </customer>
24 <customer>
25   <id>4</id>
26   <name></name>
27   <address_1></address_1>
28   <address_2></address_2>
29   <telephone_number></telephone_number>
30 </customer>
31 </customers>
32
```

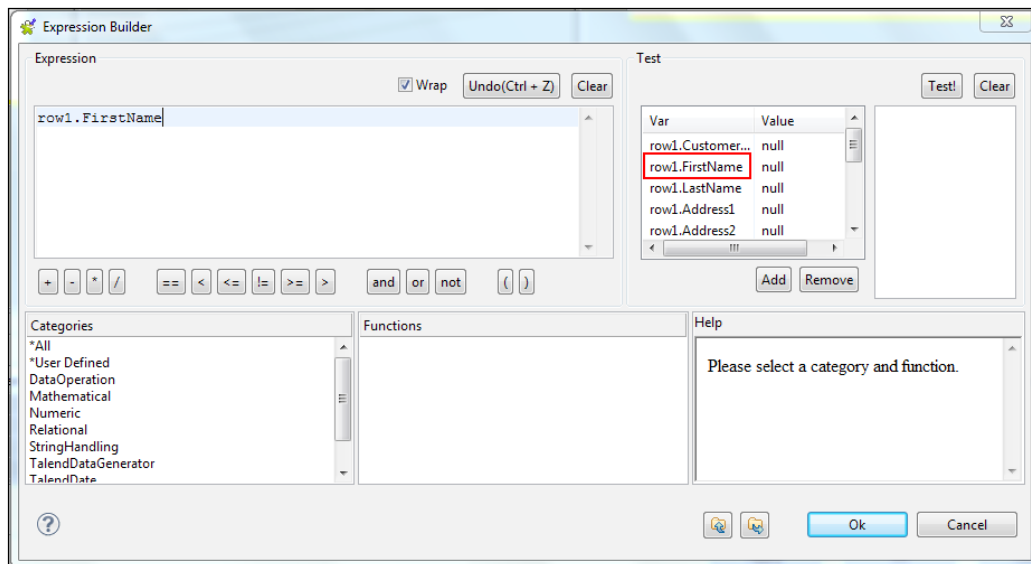
Great! It works! There are lots of empty elements, but we can soon fix that.

Open the **tMap** component again. Let's work on the **FirstName** and **LastName** fields. In this instance, we cannot perform a straight drag-and-drop function, so to start the mapping process for these fields, click on the **Expression** column of the **out1** schema on the **name** row. You will see an ellipsis button appear at the end of the line. Click on this to reveal the **Expression Builder**.



The **Expression Builder** is a really useful tool that allows you to build data transformations in Java code from a library of functions and expressions. You can also type Java code into the **Expression Builder**, if there is a Java function you want to use that is not in the function library. You'll find that you use the **Expression Builder** a lot, particularly with complex integration transformations, so take some time to practice with it. If your Java coding is a little rusty, simply Google what you want to do. There is a wealth of Java tutorials and code snippets on the Internet.

We're going to build an expression that joins the first and last names so that we can pass the output to the **name** field in the output schema. To do this, double-click on **row1.FirstName**. This will then appear in the **Expression** window:

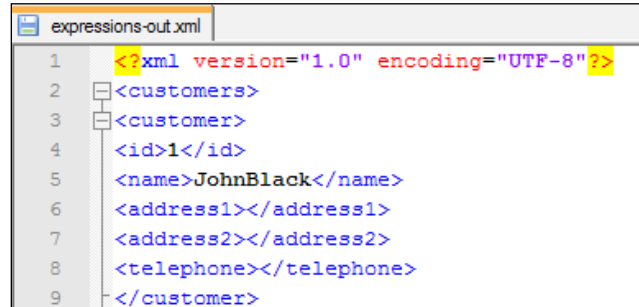


Place your cursor at the end of the expression, and then double-click on **row1.LastName**. It will appear at your cursor position, which is after **row1.FirstName**. The expression must be valid Java syntax, so we need to do a bit more work before we can test the job again. Between **row1.FirstName** and **row1.LastName** type a +, which is the Java string concatenation operator. The **Expression** window should now show the following:

```
row1.FirstName+row1.LastName
```

Click on **OK** in the **Expression Builder**, and then click on **OK** again on the mapping window. Let's see what happened, by running our job again.

Well it's definitely progress, but not quite what we had expected.

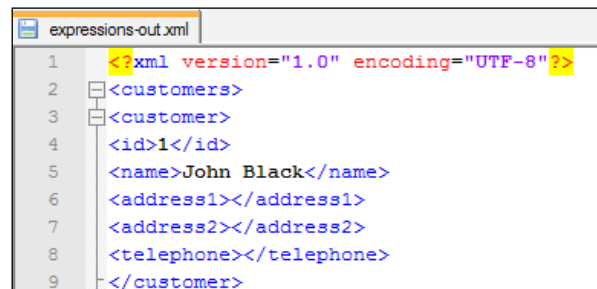


```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <customers>
3 <customer>
4 <id>1</id>
5 <name>JohnBlack</name>
6 <address1></address1>
7 <address2></address2>
8 <telephone></telephone>
9 </customer>
```

The first and last names have been joined together, but it is a continuous string, rather than the normal convention of [first name] [space] [last name]. Let's quickly resolve that by opening the **Expression Builder** again and editing the expression to the following:

```
row1.FirstName+" "+row1.LastName
```

Run the job again to check the output:



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <customers>
3 <customer>
4 <id>1</id>
5 <name>John Black</name>
6 <address1></address1>
7 <address2></address2>
8 <telephone></telephone>
9 </customer>
```

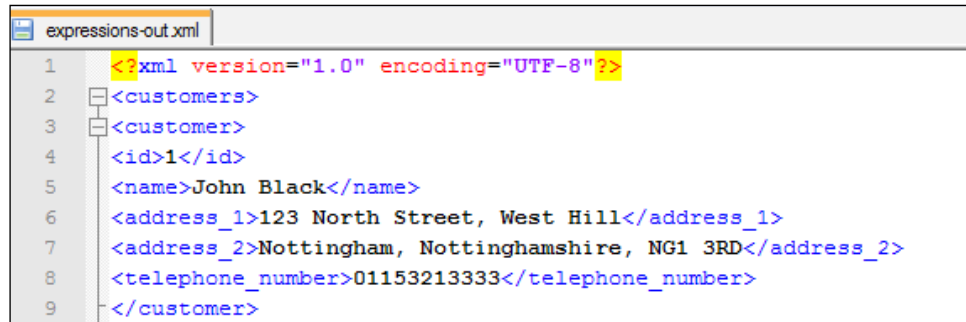
Let's complete the mapping for the other fields by creating expressions for them, as follows:

- For address_1:
`row1.Address1+", "+row1.Address2`
- For address_2:
`row1.TownCity+", "+row1.County+", "+row1.Postcode`

You can see here that we've added a comma and a space between each address field.

Let's also drag-and-drop **row1.Telephone** onto the **Expression** column of **out1.telephone_number**.

Let's run the job again. We're getting closer to our required output, but there are a few more changes that we need to make as per the requirements.



```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <customers>
3  <customer>
4    <id>1</id>
5    <name>John Black</name>
6    <address_1>123 North Street, West Hill</address_1>
7    <address_2>Nottingham, Nottinghamshire, NG1 3RD</address_2>
8    <telephone_number>01153213333</telephone_number>
9  </customer>

```

1. The `<id>` is displaying an integer, but we want this to be left-padded with zeros, up to a maximum of 8 characters. To implement this, click on the expression ellipsis button of the `id` field in the **Expression** column of the **out1** schema, and change the expression from `row1.CustomerID` to the following:

```
String.format("%08d", row1.CustomerID)
```

2. This is the Java expression for left-padding with zeros.
3. On **address_1** of the second customer record, the output is `<address_1>45 South Drive, </address_1>`. The **Address2** field in the input file is not mandatory. We get the slightly odd result of the output for the **address_1** field because its format is `[Address_1] [comma] [space]`. As we know that there will always be an **Address_1** field, but not necessarily an **Address_2** field, we can make the `[comma] [space]` output conditional, based upon the presence of an **Address_2** value. In the expression for **address1** in **out1**, enter the following:

```
("").equals(row1.Address2)?row1.Address1:row1.Address1+", "+row1.Address2
```



The previous code is the Expression Builder format for an if-then-else statement; the developers named it as the ternary condition. It takes the following form:

```
[test]?[value if true]:[value if false]
```

So in the previous code, the expression will evaluate if the **row1.Address2** field is empty. If it is empty, then the output will be as follows:

```
row1.Address1
```

If it is not empty, then the output will be as follows:

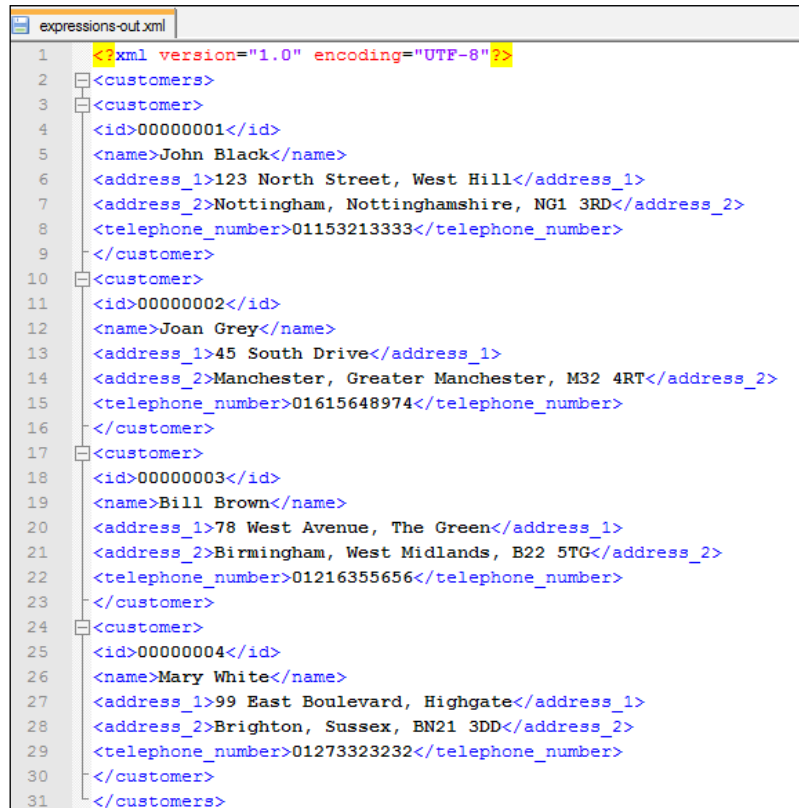
```
row1.Address1 [comma] [space] row1.Address2
```

4. We need to standardize the telephone number output and remove any spaces, hyphens, or other non-numeric characters. Let's change the expression for the **map_output telephone_number** to the following:

```
row1.Telephone.replaceAll( "[^\\d]", "" )
```

5. This is the Java expression to remove all non-numeric characters.

Let's run the job one more time to check the output. You should see something similar to the following screenshot, in the DataOut folder:



As you can see from this example, the Expression Builder is a powerful tool, allowing us to format data and add logic to our integration jobs.

We've looked at some simple file transformations and used expressions to modify data. Let's move on to look at another the Studio component called **tAdvancedFileOutputXML**, which deals with complex XML structures.

Advanced XML output for complex XML structures

The XML output format we used in the last example was pretty straightforward. Every piece of data is contained within its own XML element; however, it is very common for XML files to be more complex, with subelements being repeated within a parent element, and data items being held in XML attributes rather than in elements.

In this example, we will produce an XML file that contains data about customer orders that have been dispatched. This file will contain the following information:

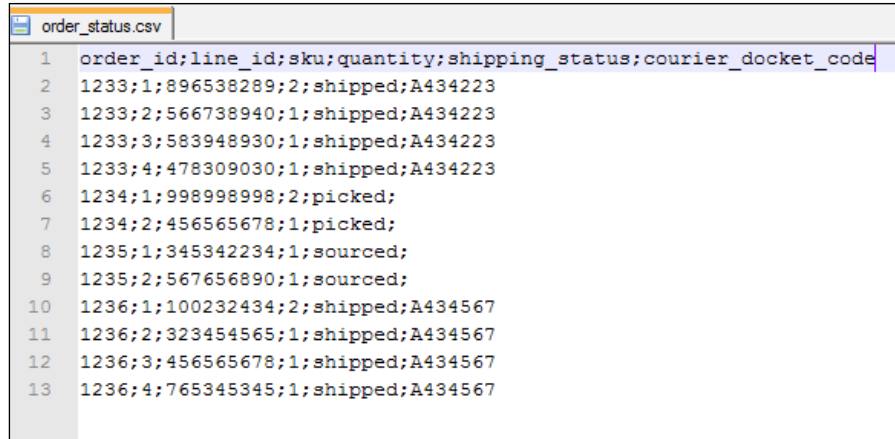
- The order ID
- The order-line ID
- The product SKU for each line
- The quantity of each dispatched SKU
- The dispatch date (in the format yyyy-MM-dd hh:mm)
- The courier tracking ID (so that customers can track their order)

The XML format we need to adhere to is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<DISPATCH_DOCKET>
  <ORDER ID="1000">
    <ORDER_LINE ID="1" SKU="123456789" QUANTITY="1" DISPATCHED_
DATE="2011-01-01 12:00" TRACKING_ID="ABC12345"/>
  </ORDER>
</DISPATCH_DOCKET>
```

Note that the XML elements, `<ORDER>` and `<ORDER_LINE>`, do not contain data between their opening and closing tags, as in the previous example; but instead, data is stored in the attributes of each of these elements. The Studio component we will use in this example is **tAdvancedFileOutputXML**. It allows us to build up complex XML structures and explicitly assign data to elements or attributes, as appropriate.

To make things a little more interesting, we are using our input CSV file with slightly different data. It contains `order_id`, `line_id`, `sku`, `quantity`, `shipping_status`, and `courier_docket_code`, so we'll have to do some manipulations to map this to the output file. An example file is shown in the following screenshot:



```
1 order_id;line_id;sku;quantity;shipping_status;courier_docket_code
2 1233;1;896538289;2;shipped;A434223
3 1233;2;566738940;1;shipped;A434223
4 1233;3;583948930;1;shipped;A434223
5 1233;4;478309030;1;shipped;A434223
6 1234;1;998998998;2;picked;
7 1234;2;456565678;1;picked;
8 1235;1;345342234;1;sourced;
9 1235;2;567656890;1;sourced;
10 1236;1;100232434;2;shipped;A434567
11 1236;2;323454565;1;shipped;A434567
12 1236;3;456565678;1;shipped;A434567
13 1236;4;765345345;1;shipped;A434567
```

Note that the input file contains data from orders in various different states, for example, shipped, sourced, and picked. Our output file is concerned only with dispatched (or shipped) orders, so we'll need to filter the data to get what we want.

Let's start building the new job as follows:

1. Create a new job and name it `AdvancedXMLOutput`.
2. We'll start, as before, by creating the metadata for the order status delimited file. Follow the same steps that we outlined previously, using the `order_status.csv` sample file as an input to the metadata definition. On step 4 of the definition, we can define our key fields. In this example, we have a composite key; `order_id` on its own is not a unique key, but is the combination of `order_id` and `line_id`. It does provide a unique reference for each row, so make both the columns key fields, as shown in the following screenshot :

New Delimited File

File - Step 4 of 4

Add a Schema on repository
Define the Schema

Name:

Comment:

Schema

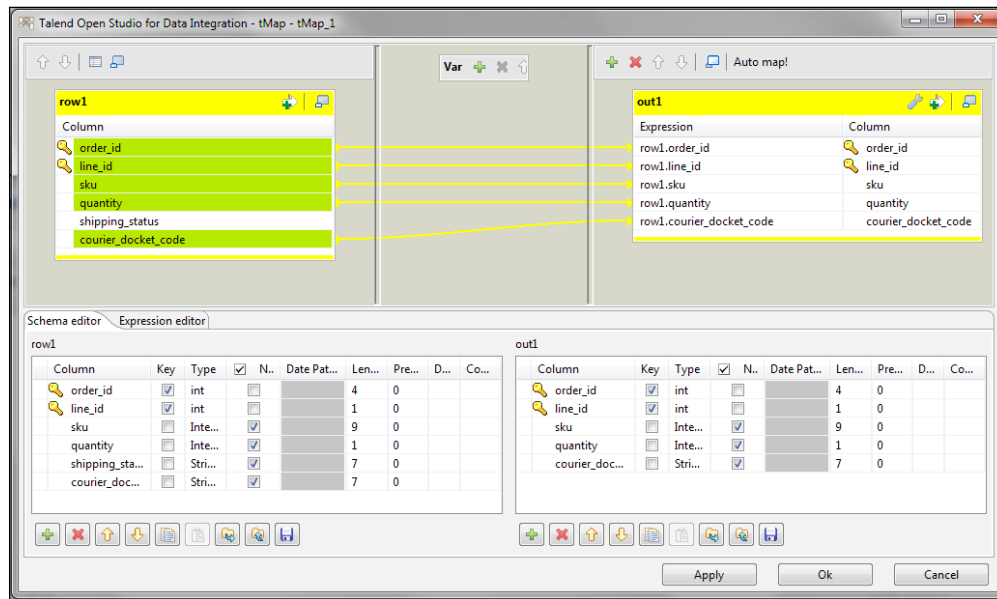
Click to update schema preview

Description of the Schema

Column	Key	Type	<input checked="" type="checkbox"/>	Nullable	Date Pattern (Ctrl...	Length	Precision	Default
order_id	<input checked="" type="checkbox"/>	int	<input checked="" type="checkbox"/>	<input type="checkbox"/>		4	0	
line_id	<input checked="" type="checkbox"/>	int	<input checked="" type="checkbox"/>	<input type="checkbox"/>		1	0	
sku	<input type="checkbox"/>	Integer	<input checked="" type="checkbox"/>	<input type="checkbox"/>		9	0	
quantity	<input type="checkbox"/>	Integer	<input checked="" type="checkbox"/>	<input type="checkbox"/>		1	0	
shipping_status	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>		7	0	
courier_docket_code	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>		7	0	

- When you have completed the metadata setup, drag-and-drop an instance of the order status metadata onto the Job Designer. Select a **tFileDelimitedInput** component from the pop-up window.
- From the **Palette** window, search for and add a **tMap** and **tAdvancedFileOutputXML** component to the Job Designer.
- Right-click on the input delimited file, select **Row | Main**, and connect this to the **tMap** component. Double-click on the **tMap** to open the Map Editor.

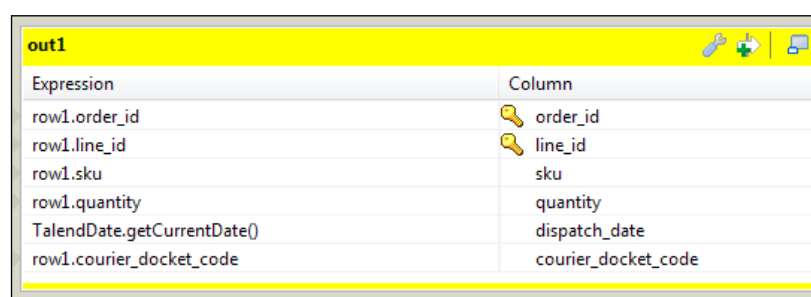
- Click on the **+** button at the top of the output pane to add a new data output. We now need to define the output schema and mappings we require. As most of the output elements are the same as the input elements, we can drag-and-drop these from the input pane on the left-hand side, to the output pane on the right-hand side. Multiselect **order_id**, **line_id**, **sku**, **quantity**, and **courier_docket_code** from the input panel, and drag them over to the newly created output pane:



- We also need the dispatch date in the output schema, so click on the **+** button in the **out1** pane of the **Schema editor** tab, to add a new column to the schema and change its name to **dispatch_date**. Change its data type to **Date** and amend the **Date Pattern** field to **yyyy-MM-dd hh:mm**. For consistency with the element order of the output file, let's also use the up arrow button to move the date above the **courier_docket_code** column.

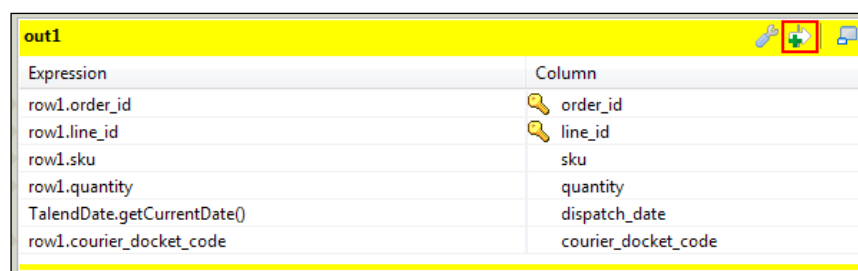
Column	Key	Type	✓	N..	Date Pattern (Ctrl+Space availab...
order_id	<input checked="" type="checkbox"/>	int	<input checked="" type="checkbox"/>		
line_id	<input checked="" type="checkbox"/>	int	<input checked="" type="checkbox"/>		
sku	<input checked="" type="checkbox"/>	Integer	<input checked="" type="checkbox"/>		
quantity	<input checked="" type="checkbox"/>	Integer	<input checked="" type="checkbox"/>		
dispatch_date	<input checked="" type="checkbox"/>	Date	<input checked="" type="checkbox"/>		"yyyy-MM-dd hh:mm"
courier_doc...	<input checked="" type="checkbox"/>	String	<input checked="" type="checkbox"/>		

8. The output file requires a dispatch date, but this data is not contained in the input file. In the real world, we might have a number of choices at this point. For example, we could change the input file to include a dispatch date so that it can be mapped to the output file. In our case, we will generate a relevant date using the Studio date functionality. Click on the ellipsis in the **Expression** column of the dispatch date, which will bring up the Expression Builder. In the **Categories** section in the bottom left-hand corner, scroll through and click on **TalendDate**. This will show a number of date functions in the middle **Functions** window. Find the function **getCurrentDate** and double-click on it. The function syntax will appear in the expression code window. Click on **OK** in the Expression Builder window to save the changes.



Expression	Column
row1.order_id	order_id
row1.line_id	line_id
row1.sku	sku
row1.quantity	quantity
TalendDate.getCurrentDate()	dispatch_date
row1.courier_docket_code	courier_docket_code

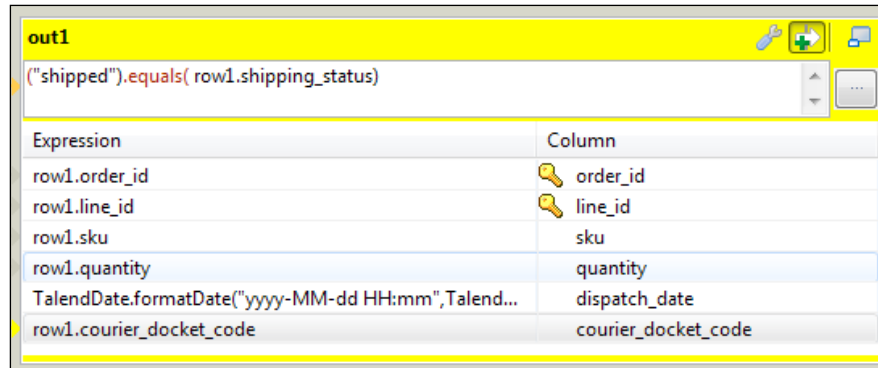
9. As noted previously, our input file contains the data for orders that are yet to be dispatched, so we want to remove this data from the output. The **tMap** component contains an expression filter that allows us to do just that. In the top right-hand corner of the **out1** schema is a white arrow icon with a green + sign (highlighted in red in the following screenshot):



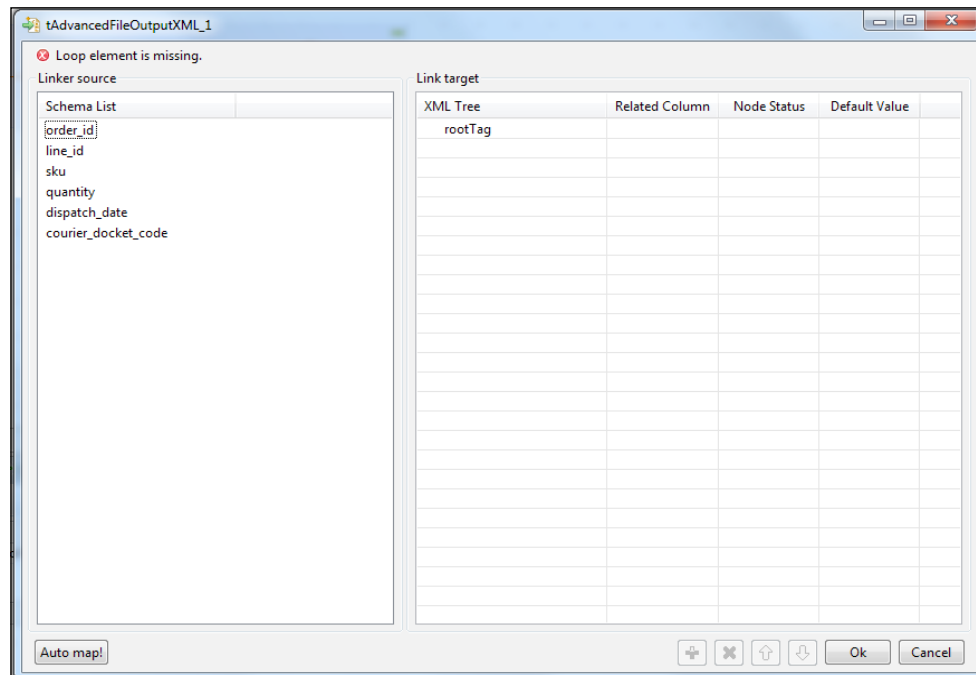
Expression	Column
row1.order_id	order_id
row1.line_id	line_id
row1.sku	sku
row1.quantity	quantity
TalendDate.getCurrentDate()	dispatch_date
row1.courier_docket_code	courier_docket_code

10. Click on this to reveal the expression filter editor. We want to filter our data to where the shipping status on the input file is shipped, rather than when it is picked, sourced, or anything else. To implement this, drag the **shipping_status** field from the input schema across to the expression filter box. It will show as **row1.shipping_status**. Edit this so that it shows as the following:
- ```
("shipped").equals(row1.shipping_status)
```

11. This will only allow rows with **shipping\_status** of "shipped" through to the **out1** schema. Click on **OK** in the **tMap** component to save the changes.

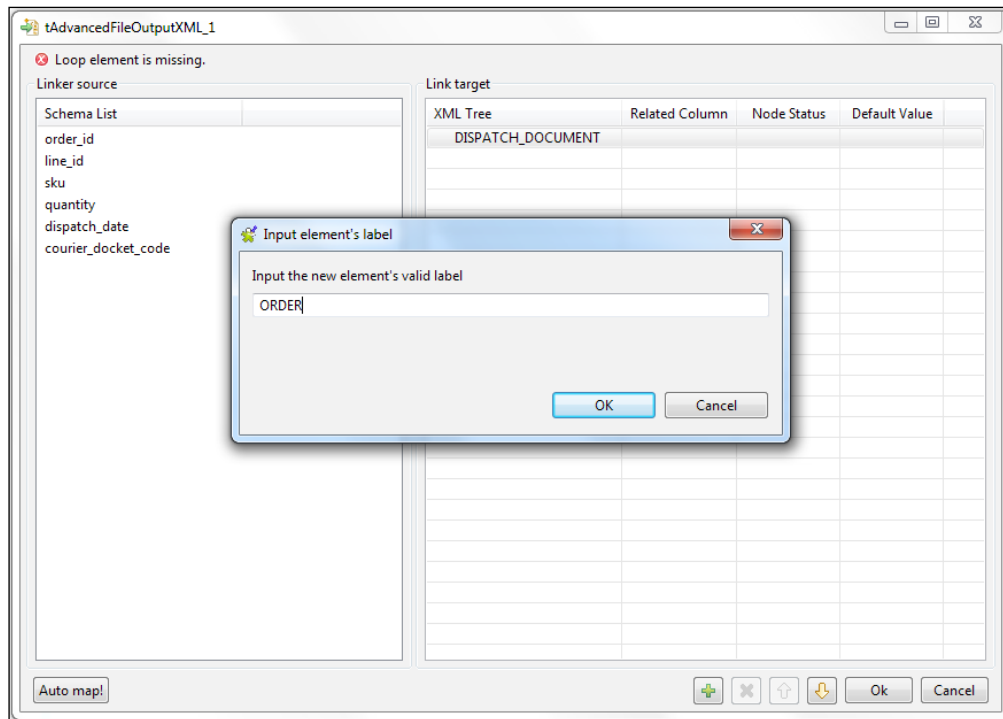


12. We now need to configure the XML output. Click on the **tAdvancedFileOutputXML** component. In the **Basic settings** tab, click on the **Configure Xml Tree** button. In the window that pops up, you can see our schema in the left-hand pane and an empty **XML Tree** window on the right-hand side. Our job here is to construct the **XML Tree** column and then assign data elements from the left-hand window to this tree.



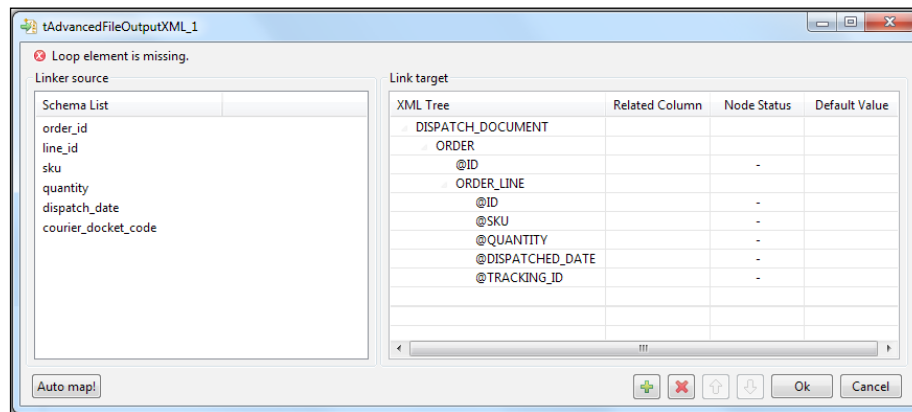


13. We'll start with the **rootTag** field. From our example output file, we can see that the root tag is `<DISPATCH_DOCUMENT>`. Click on the **rootTag** in the **XML Tree** column and change its value to **DISPATCH\_DOCUMENT**. The `<ORDER>` element is a subelement of `<DISPATCH_DOCUMENT>`, and `<ORDER_LINE>` is a subelement of `<ORDER>`. Right-click on **DISPATCH\_DOCUMENT** in the **XML Tree** column and select **Add Sub-element**. Enter **ORDER** in the pop-up window and click on **OK**:

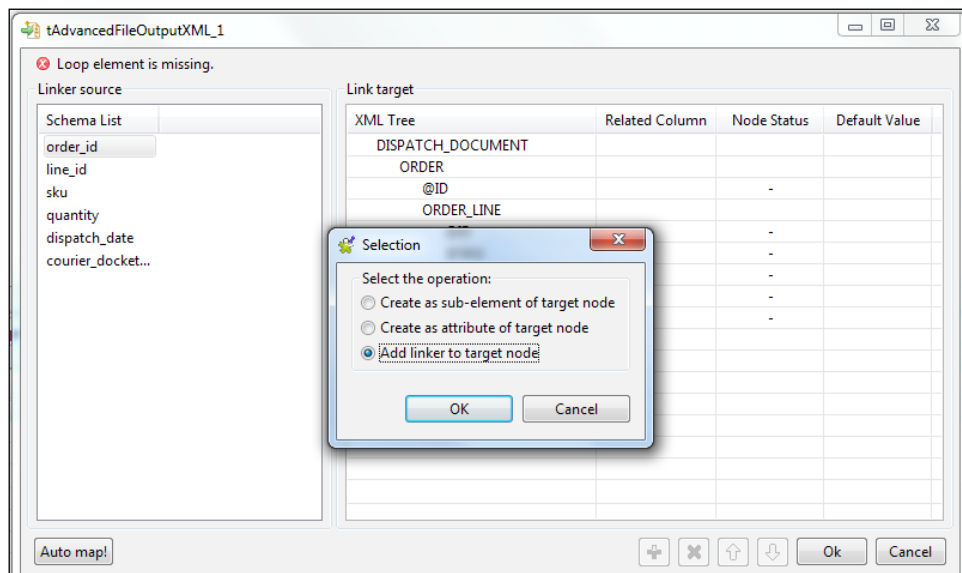


14. Follow the same process for creating the subelement **ORDER\_LINE**. We can now create the attributes.
15. Right-click on **ORDER** and select **Add Attribute**. In the pop-up window, enter the attribute value **ID** and click on **OK**. You'll see the attribute appear below the **ORDER** field in the **XML Tree** column (attributes are prefixed with the @ symbol).

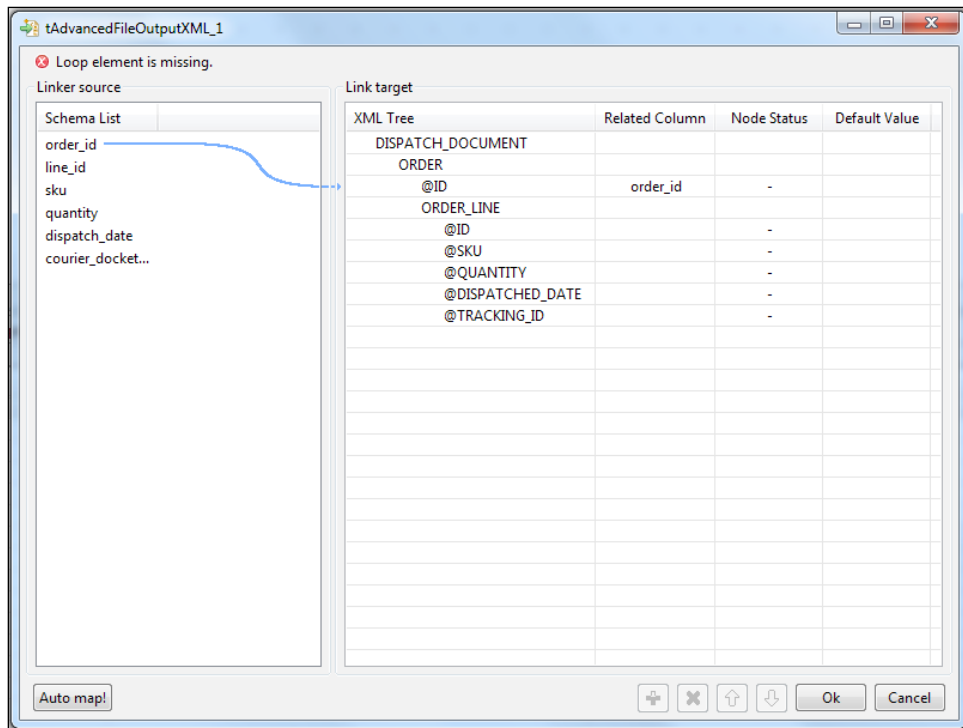
16. On the **ORDER\_LINE** element, we need to add **ID**, **SKU**, **QUANTITY**, **DISPATCHED\_DATE**, and **TRACKING\_ID** as attributes, so go ahead and do this by following the procedure outlined previously. Your **XML Tree** will look similar to what is shown in the following screenshot, when you have finished:



17. Now let's connect our data inputs with the new **XML Tree**. As we saw with the **tMap** component, linking can be achieved by dragging-and-dropping. Click on the **order\_id** item in the left-hand pane and drag it onto the **@ID** element of the **XML Tree** tab. You will be presented with a pop-up window that offers a number of linking options:

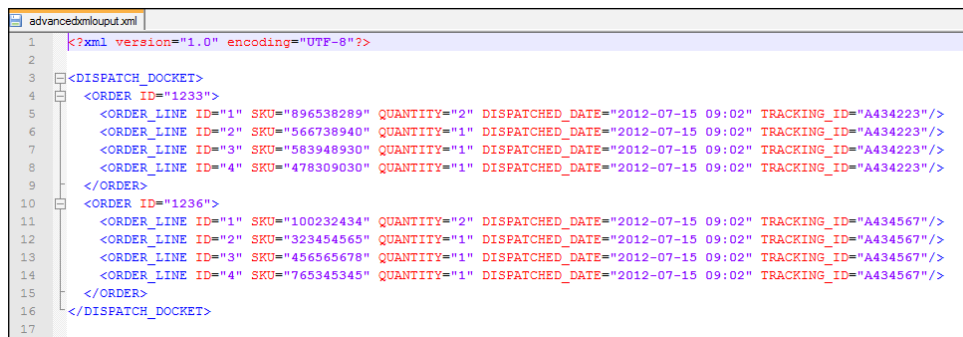


18. The first two options allow you to assign data items to and create the XML tree in one go. Alternatively, you can use the last option, which links the data items to an already existing XML tree. Try both methods, as you develop your integration jobs, and see which works best for you. As we have already created our XML tree, we'll use the last option, which is **Add linker to target node**. Click on **OK**, and the Studio will draw a connection line between the data item and **XML Tree**:



19. In the same way, connect **line\_id** to **@ID**, **sku** to **@SKU**, **quantity** to **@QUANTITY**, **dispatch\_date** to **@DISPATCHED\_DATE**, and **courier\_docket\_code** to **@TRACKING\_ID**.
20. You will notice that the XML tree builder is showing a warning in the top left-hand corner of the window, **Loop element is missing**. We need to specify the element that we want the the process to loop over. This will typically be the element that defines a row. In our case, our rows are based on order lines, so we will use the **ORDER\_LINE** element of the **XML Tree** tab as our loop element. Right-click on **ORDER\_LINE** in the XML tree and select **Set As Loop Element**.

21. We can also optionally set a **group by** element. This is the element that the loop element is grouped by. For some XML outputs this may not make sense, but in our output, we are outputting the order lines and we want to group them by order. Right-click on **ORDER** in the XML tree and select **Set As Group Element**. Click on **OK** to accept the changes.
22. Amend the **File Name** parameter under the **Basic settings** tab, and change the **Encoding** value to **UTF-8** on the **Advanced settings** tab.
23. Run the job and check the output. You should see something similar to the following screenshot:



You can see that the Studio's advanced XML output gives us complete flexibility to build files to a given specification.

## Working with multi-schema XML files

The XML files we have worked with so far were straightforward and only dealt with a single collection of elements. However, many systems produce or consume XML files that contain multiple collections of elements; these are called multi-schema XML files. Here's an example:

```
<?xml version="1.0" encoding="UTF-8"?>

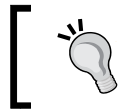
<catalogue>
 <skus>
 <sku>
 <skuid>432345</skuid>
 <skuname>Check Shirt</skuname>
 <size>S</size>
 <colour>Green</colour>
 <price>29.99</price>
 </sku>
 </skus>
</inventory>
```

```

 <sku>
 <skuid>432345</skuid>
 <stock_on_hand>12</stock_on_hand>
 </sku>
 </inventory>
</catalogue>

```

This shows a product catalogue file with two schemas, one for the product details and one for the inventory. There's nothing in the XML structure to connect the two schemas. In essence, these are two separate XML structures joined together for convenience. In order to process this file in the Studio, we could simply create two XML file input components, and have one input read the SKUs schema and the other read the inventory schema. However, the Studio makes things a little easier for us, by offering a multi-schema XML file input component. We'll create a job that reads the XML files and writes the output to the **Run job** console using the **tLogRow** component.

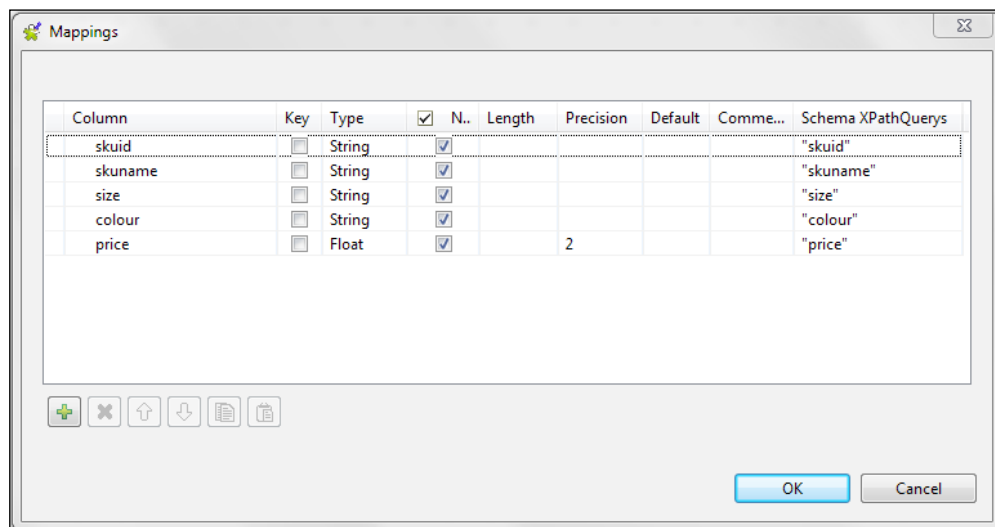


The **tLogRow** component is great for quickly testing the output of a job before writing it into a more complex structure, such as an XML file, or before loading it into a database.

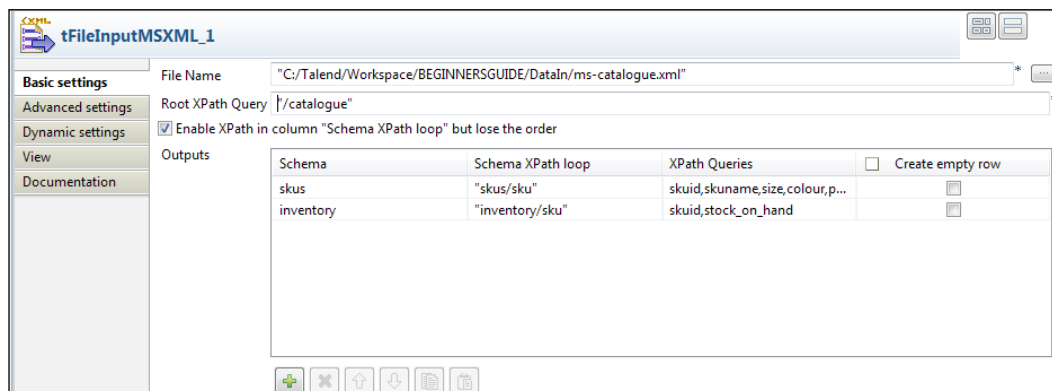
We'll use the input file, `ms-catalogue.xml`, from the Chapter 3 example files as our input data. Copy this to your `DataIn` folder. Follow these steps to create the job:

1. Create a new job and name it `MultiSchema`. The Studio does not offer a metadata component for a multi-schema XML file, so in this case, we will configure directly in the component.
2. In the **Palette** window, search for `MSXML` (multi-schema XML), and from **File | Input**, drag-and-drop a **tFileInputMSXML** component onto the Job Designer.
3. Now search for `logrow` in the **Palette** window and drop two **tLogRow** components onto the Job Designer.
4. Click on the **tFileInputMSXML** component and configure its properties.
5. Set the **File Name** to the datafile in your `DataIn` folder.
6. Next we'll set the **Root XPath Query** field, which in our case will be `/catalogue`.
7. Click on the checkbox to **enable XPath in the column**. This will allow us to define an XPath loop for each schema.
8. In the **Outputs** section, click the **+** button to add a schema.
9. In the schema column of the newly created row, click on the ellipsis button. You'll get a pop-up box where you can define the name of the schema; let's call this **SKUs**.

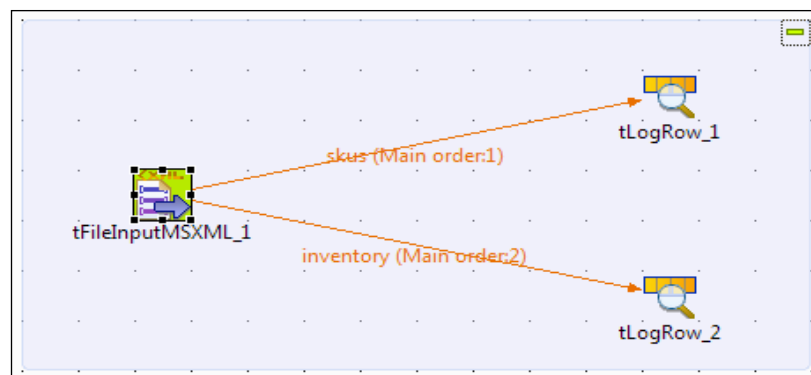
10. A follow-on pop-up window will allow you to define the schema components for the **skus** schema. Click the **+** button five times and add the following elements:
  - skuid
  - skuname
  - size
  - colour
  - price
11. Leave the data types as **String**, except for price, which we will change to **Float**, with **Precision** as **2**. Click on **OK** to save the schema.
12. In the **Schema XPath Loop** column, enter the XPath loop for the **skus** schema. Note that this is relative to the **Root XPath Query** field that we configured in step 6 previously, so we'll set this to **skus/sku**.
13. We now need to set the XPath queries for the individual fields. Click on the ellipsis in the **XPath Queries** column and enter the following values in the **Schema XPathQueries** column of the pop-up window:
  - sku
  - skuname
  - size
  - colour
  - price



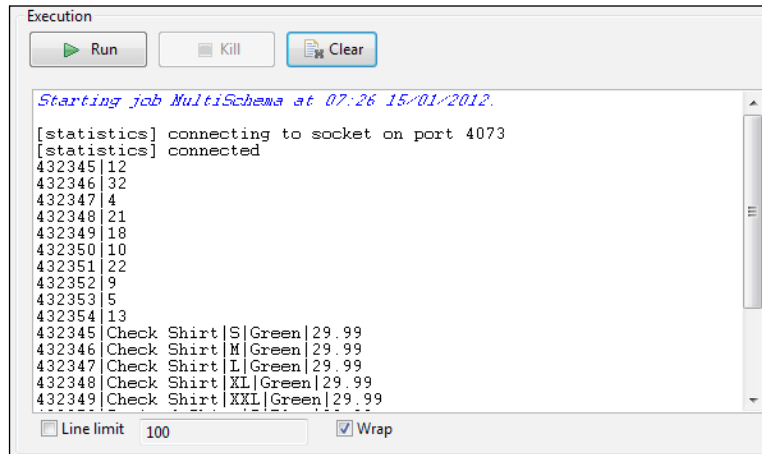
14. Follow steps 8 to 12 to configure the component for the inventory schema.



15. Let's now connect our multi-schema XML component to the log rows. Right-click on the **tFileInputMSXML** component, select **Row | skus**, and drop the connector onto **tLogRow1**. Right-click on the **MSXML** component again, select **Row | Inventory**, and drop the connector onto **tLogRow2**:



16. We're now ready to run our job. Go to the **Run** tab in the bottom panel and click on **Run**. You will see the data scroll past in the **Run job** console. You will see the inventory data returned, followed by the product data:



Nice work! Obviously, we could send the output to different files rather than the **Run Job** console if required.

## Enriching data with lookups

So far, we have looked at integration scenarios where we have transformed files from one format to another, but in all cases the data we needed in the output file was contained, in some form, in the input file. However, it is commonplace in real-life scenarios that we need to transform data to the requirements of one system, but the originating system does not actually contain the data we need. It's time to improvise!

In this section, we'll create a job that passes data from one component to another, but on the way, uses a lookup data to replace some data. Imagine that we need to transform some customer data. Our original file is simple, containing the following fields:

- Company name
- Address
- City
- State
- Zip code

The following is a sample file:



corporate-addresses.csv				
1	company_name;address;city;state;zip			
2	Talend;5150 El Camino Real;Los Altos;California;94022			
3	Microsoft;One Microsoft Way;Redmond;Washington;98052			
4	Apple;1 Infinite Loop;Cupertino;California;95014			
5	eBay;2065 Hamilton Avenue;San Jose;California;95125			
6	Dell;One Dell Way;Round Rock;Texas;78682			
7	IBM;1 New Orchard Road;Armonk;New York;10504			

Let's name this file as `corporate-addresses.csv` and drop it into your `DataIn` folder.

The output file required by the receiving system is exactly the same as this, with one exception. Its **state** field is only two characters long (as it is expecting the standard USPS two-character state code).

Our original system does not hold the two-character state code, so we'll have to get this data from somewhere else and merge it into the original file. In this case, a quick search on the Internet will direct us to a number of sources of state name and state code data. For this job, we're going to keep the lookup data in a pretty low-tech container. Fire up your trusted text editor and write the state data into the file, in the following format:

```
[State Name], [State Code]
```

Make sure that each state name/state code pair is on a new line. The following is a snippet of what you should have when you have completed this:

states.csv	
1	state_name,state_code
2	Alabama,AL
3	Alaska,AK
4	Arizona,AZ
5	Arkansas,AR
6	California,CA
7	Colorado,CO
8	Connecticut,CT
9	Delaware,DE
10	District of Columbia,DC
11	Florida,FL
12	Georgia,GA
13	Hawaii,HI
14	Idaho,ID
15	Illinois,IL
16	Indiana,IN
17	Iowa,IA
18	Kansas,KS
19	Kentucky,KY
20	Louisiana,LA
21	Maine,ME

We'll name this file as `states.csv` and drop it into the `DataIn` folder too.

Of course, the state lookup data could be held in lots of different containers, such as an XML file, relational database, even a third-party web service; but for now we'll work with a plain old text file.

Let's start by creating a new job named `StateLookup`. Then, follow these steps:

1. Create file delimited metadata for the two input files, `corporate-addresses.csv` and `states.csv`, following the same steps as used previously.

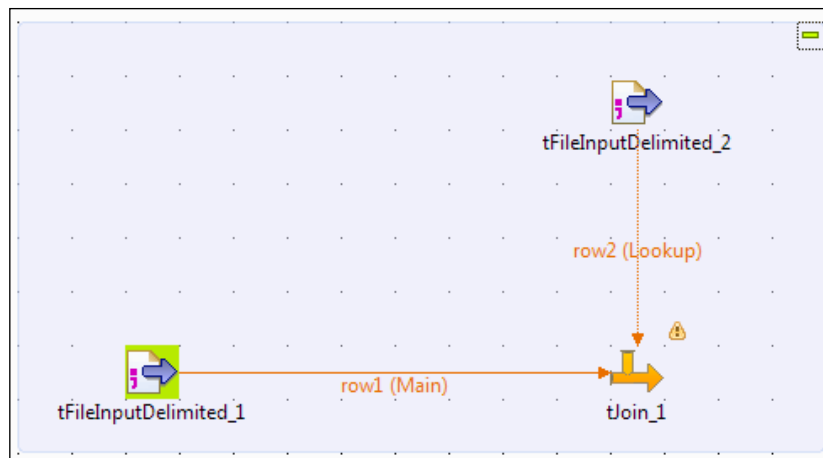


When creating the corporate addresses metadata, the wizard defines `zip` as a data type of integer, based on the data presented in the file. This is okay if the file contains US-only addresses, but might not be okay if the file contains a mix of US and other addresses. If this were to be the case, you would modify the data type here to string, to cope with other zip code formats.

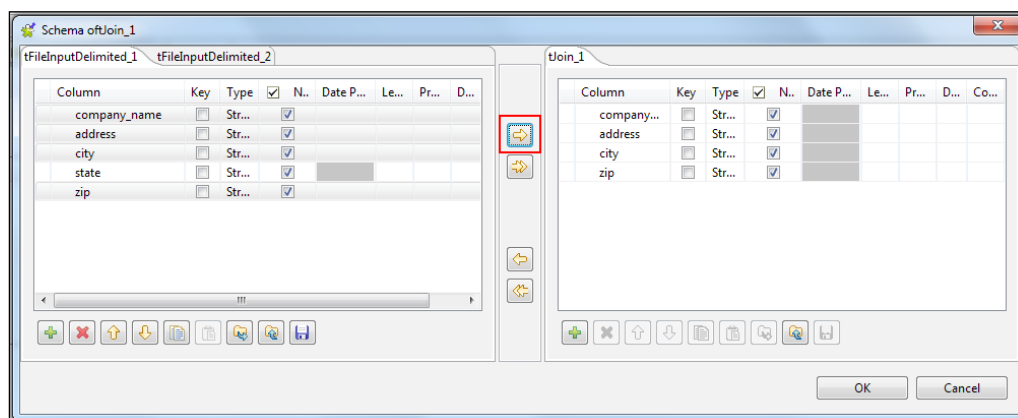
2. Drag each metadata component onto the Job Designer, selecting **tFileInputDelimited** in each case.
3. Now search for `join` in the **Palette** window, grab a **tJoin** component, and drop it onto the designer. This is the component that will join our two data streams.
4. Right-click on the corporate addresses file component, select **Row | Main**, and drop the connector onto the **tJoin** component.
5. Right-click on the `states` file component, select **Row | Main**, and again drop this onto the **tJoin** component. You'll notice that, even though we selected **Row | Main** in both cases, the resulting connector is slightly different. The first connector we made in step 4 is noted as **Main**, while the second is called **Lookup**. The order in which we connect the components to the **tJoin** is important. Join the primary data stream first, and one or more lookup components subsequently.



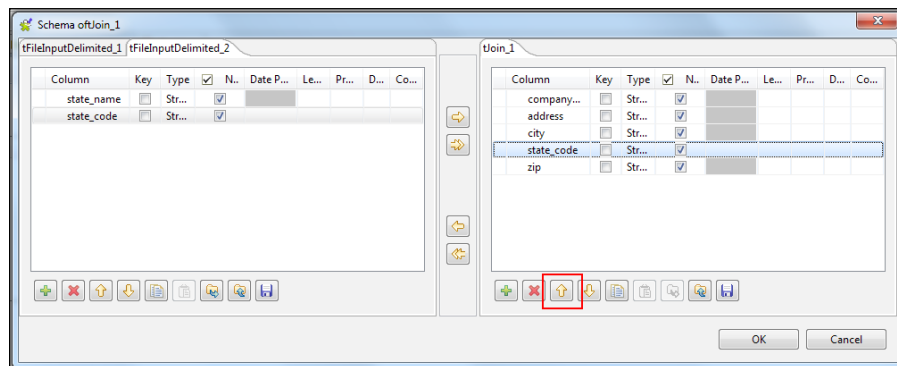
If you join connections in the wrong order or wish to change which is the Main dataflow and which is the Lookup dataflow, you can do this by right-clicking on one of the connectors and selecting **Set this connection as Lookup** or **Set this connection as Main**.



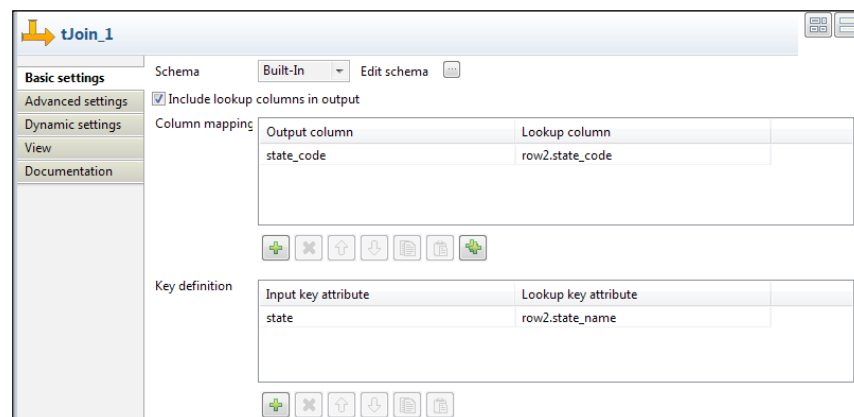
6. Search for delimited in the **Palette** window and drag a **tFileOutputDelimited** component onto the Job Designer. Edit its **File Name** so that the output is created in your DataOut folder. Let's call the output file address-lookup-out.csv.
7. Now click on the **tJoin** component. In its **Basic settings**, click on the **Edit Schema** button.
8. We need to copy some of the input fields into the **tJoin** component. You can see in the left-hand side window that both of the input schemas are shown in separate tabs. Holding down the **Ctrl** key, click on **company\_name**, **address**, **city**, and **zip** from our main delimited input file. Click on the top-arrow button in the middle bar of the window to copy these over to the **tJoin\_1** schema. (Clicking on the two-arrow button will copy all the fields from the left-hand side window to the right-hand side window.)



9. Now click on the second tab in the left-hand side window showing the schema for the second delimited file input. Select the **state\_code** column and copy this over to the right-hand side window, using the top-arrow button.
10. In the right-hand side window, click on the **state\_code** column and move it up one place on the list of fields by clicking on the up arrow button:



11. Click on **OK** to save the changes, and when prompted to propagate the schema changes, click on **Yes**. This will copy the **tJoin** schema over to the delimited output component.
12. In the **Basic settings** tab of the **tJoin** component, click on the checkbox **Include lookup columns in output**.
13. In the **Column mapping** box, add a row and select **state\_code** from the **Output column** drop-down menu. In the **Lookup column** section, select **row2.state\_code**.
14. In the **Key definition** table, add a row and select **state** as the **Input key attribute** field, and **row2.state\_name** as the **Lookup key attribute** field;





What have we just done here? In the **Key definition** table, we've noted that the **state** field in the main input file maps to the **state\_name** field in the lookup file. We've also configured in the **Column mapping** table, that in the **Output column** section, the **state\_code** field should be populated with values from the lookup file **state\_code** field.

15. Right-click on the **tJoin** component, select **Row | Main**, and drop this onto the delimited output component.
16. It's time now to run the job. You should get the following output in your DataOut folder:

```

address-lookup-out.csv
1 Talend;5150 El Camino Real;Los Altos;CA;94022
2 Microsoft;One Microsoft Way;Redmond;WA;98052
3 Apple;1 Infinite Loop;Cupertino;CA;95014
4 eBay;2065 Hamilton Avenue;San Jose;CA;95125
5 Dell;One Dell Way;Round Rock;TX;78682
6 IBM;1 New Orchard Road;Armonk;NY;10504
7

```

Notice that the **state** field has now changed from the full state name to the two-character state code.

For the final exercise in this chapter, let's build some jobs that use the most familiar of datafiles—Excel.

## Extracting data from Excel files

Spreadsheets are a ubiquitous business tool in the modern world, and there is a vast amount of critical data that resides in these common desktop files. As the tools are so commonplace and easy to use, spreadsheets are often the tool of choice for storing and manipulating all kinds of data. In this section, we'll look at a couple of ways to pull data from a spreadsheet. One of the most-used features of spreadsheets is the sheets functionality, which is the ability to add another page within the spreadsheet file. Sheets within a file may be closely related (for example, each sheet represents sales data for a given month) or may be less closely related (for example, a customers spreadsheet may contain customer data, such as the first name, last name, and e-mail in sheet 1, and address data in sheet 2). Instead of taking spreadsheet data and converting it into the CSV format before transforming it, the Studio has Excel components that allow us to address multiple spreadsheets within a single file.

## Extracting data from multiple sheets

In the first example, let's look at extracting data from multiple sheets within a spreadsheet, but where each sheet has the same columns or schema. Our example datafile has three sheets representing categories of garment. Each sheet has a **product\_code** field and a **product\_name** field. Our task is to extract data from each sheet and export it into a single CSV file. Perform the following steps:

1. Create a new job named `Spreadsheet1`.
2. As is our common practice, let's create a metadata component for our input Excel file. Right-click on **File Excel** in the **Metadata** section of the **Repository** window and select **Create file Excel**.
3. In step 1 of the wizard, change the **Name** field to `products` and click on **Next**.
4. In step 2, browse to the `products.xls` file in the `DataIn` folder. The wizard will show the available sheets in the Excel file and a preview of the data for each sheet.
5. Check **All Sheets** in the **Set sheets parameters** pane to configure all the sheets to be read by the job. In the lower right-hand pane, we need to select a sheet that will serve as a schema guide for the wizard. Let's leave this as **dresses**, as selected by the wizard. Click on **Next**.
6. In step 3, let's set the heading row as the column name, as we have done previously. Note that the heading rows from the second and third sheets are showing the preview as data, as shown in the next screenshot. Don't worry about this for now; we'll return to it when we configure the job.

**New Excel File**

**File - Step 3 of 4**  
Add a Metadata File on repository  
Define the setting of the parse job

**File Settings**

Encoding: **UTF-8**

☐ Advanced separator(for number)

Thousands separator: ,

**Metadata column setting**

First column: 1

Last column:

**Rows To Skip**

If any rows must be ignored, specify the following parameters

Header ☒ 1

Footer ☐

**Limit Of Rows**

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

Limit ☐

**Preview** | **Output**

☒ Set heading row as column names **Refresh Preview**

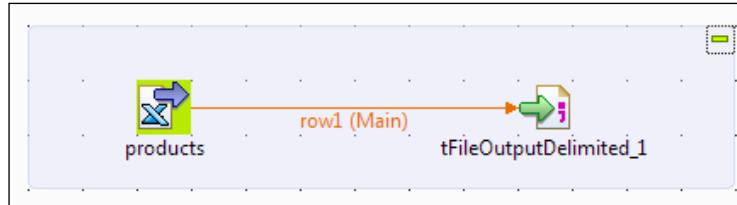
product_id	product_name
12345678	Red Dress
12345679	Pink Dress
12345680	Blue Dress
12345681	Green Dress
12345682	Black Dress
product_id	product_name

**Export as context** **Revert Context**

**< Back** **Next >** **Finish** **Cancel**

- Click on **Next**. Change the schema name to `products` in step 4 and click on **Finish**.
- Now drag the products' Excel metadata onto the Job Designer. Click on the **Component** tab below the Job Designer and check the checkbox named **Affect each sheet (header & footer)**. This forces the header configuration to apply to each sheet and resolves the issue we noted earlier, that the header data was appearing in the data output.
- Search for `delimited` in the **Palette** window and drag a **tFileOutputDelimited** component onto the Job Designer.

- Right-click on the Excel component, select **Row | Main**, and connect it to the delimited output component:



- Turning to the delimited output component, set the output **File Name** to `product-excel-out.csv` in your `DataOut` folder.
- Click on the **Include Header** checkbox, which will add the header row to the output file.
- Run the job and check the output; it should look like the output shown in the following screenshot:

The screenshot shows a text editor window with the title 'product-excel-out.csv'. The content is a CSV file with 16 lines. Line 1 is the header: 'product\_code;product\_name'. Lines 2-15 contain product data, and line 16 is empty.

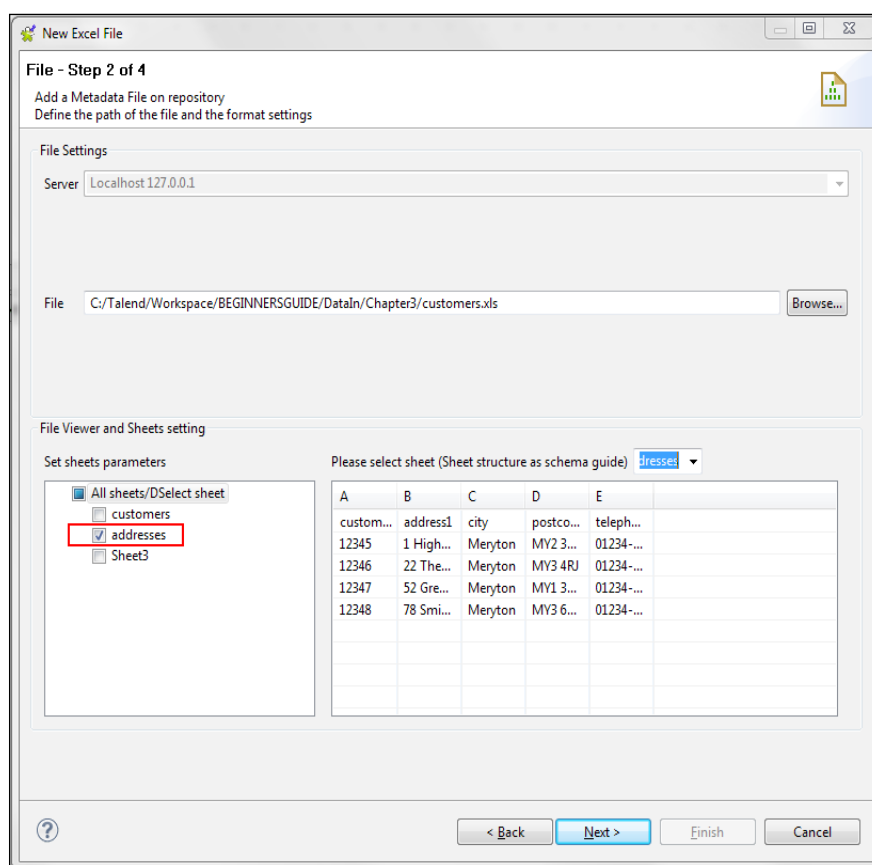
Line	product_code;product_name
2	12345678;Red Dress
3	12345679;Pink Dress
4	12345680;Blue Dress
5	12345681;Green Dress
6	12345682;Black Dress
7	98765432;Blue Striped Skirt
8	98765431;Pink Spotted Skirt
9	98765430;Multi-coloured Skirt
10	98765429;Yellow Check Skirt
11	98765428;Black Skirt
12	23456789;Blue Cardigan
13	23456790;Pink Jumper
14	23456791;Green Cardigan
15	23456792;Black Turtle Neck
16	



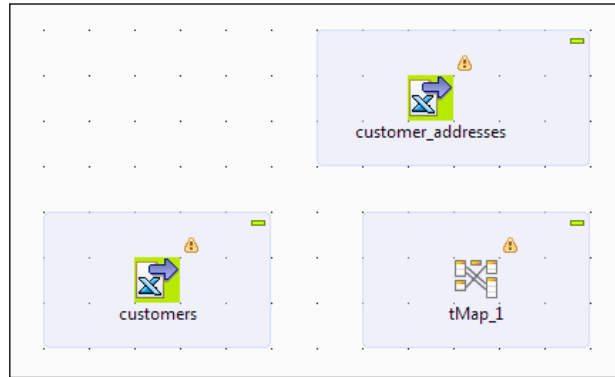
## Joining data from multiple sheets

In the second example, let's imagine our spreadsheet has two sheets—one that contains basic customer information, for example, title, name, e-mail; and another that contains customer address data. The two sheets are linked by a unique customer ID. We want to extract the data from both these sheets, join it together based on the unique customer id, and present it as a single output file. We'll use two different Excel input components to access the different sheets (and the schemas that they contain) and a **tMap** component to join the two dataflows. Perform the following steps:

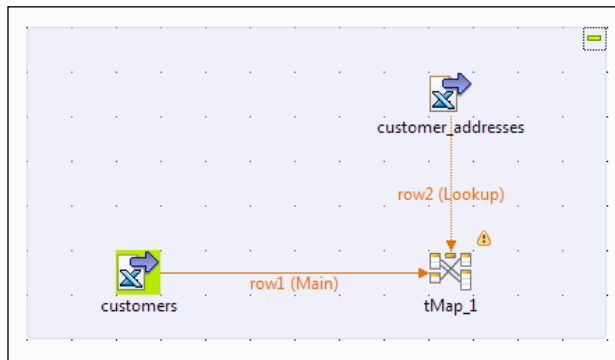
1. Create a new job named `Spreadsheet2`.
2. Let's now create two metadata components from different worksheets in the same Excel spreadsheet (`customers.xml` in the `DataIn` folder). Follow the same steps that we have used previously, but on step 2 of the wizard, ensure that the correct worksheet is checked, rather than **All sheets**. In the following screenshot, you can see just the addresses sheet being configured:



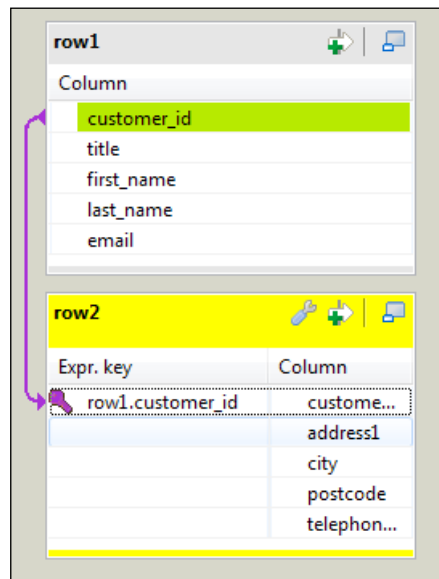
3. Once the metadata is configured, drop an instance of each from the **Metadata** section in the **Repository** window, to the Job Designer. From the **Palette** window, search for tMap, and drop this onto the Job Designer too:



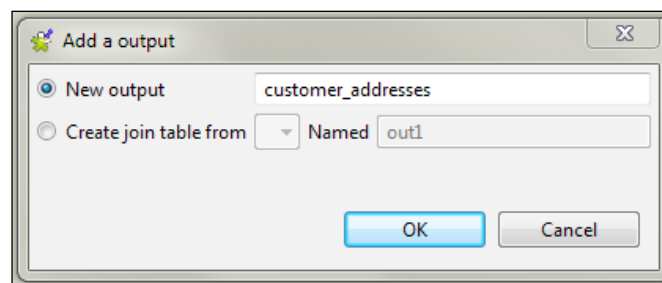
4. Now let's join everything together. Right-click on the `customers` input file, select **Row | Main**, and drop the connector onto the `tMap` component. Do the same with the `customer_addresses` input file. Note that we see the `customers` connector as **Main** and the `customer_addresses` connector as **Lookup** again. Remember that the first connection becomes the main flow and the second or subsequent connections become lookup flows:



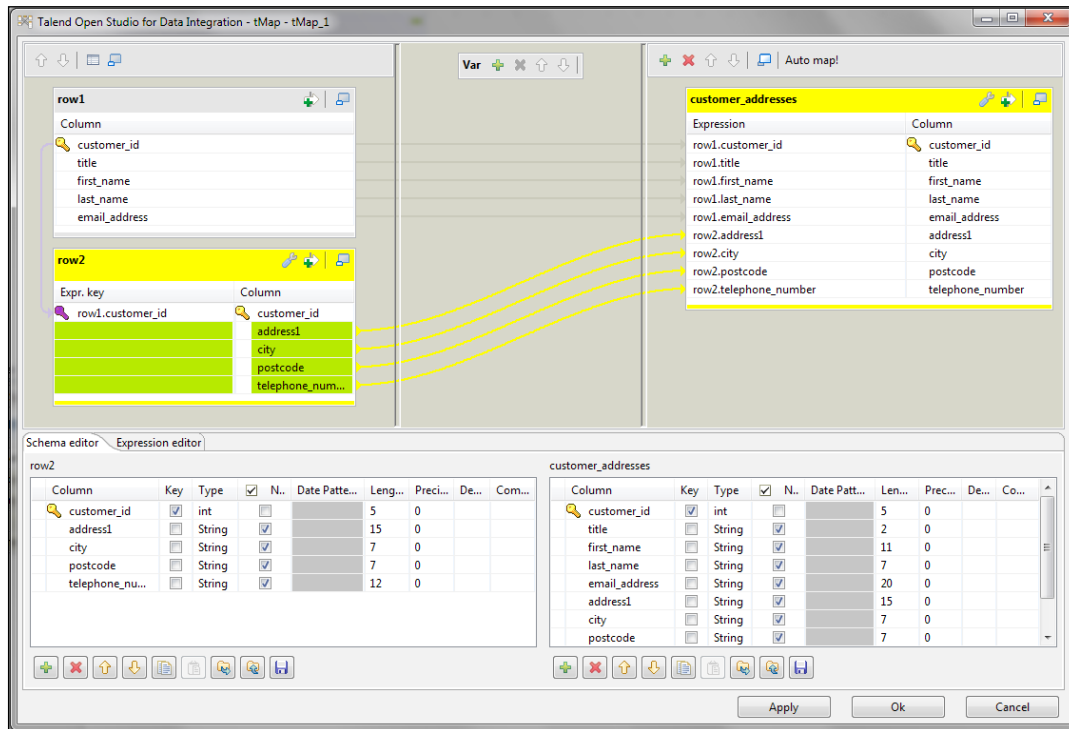
- Double-click on the **tMap** component to open the Map Editor. The first thing we need to do is join the two input rows. We have previously noted that **customer\_id** is the common field and the key that joins the two sets of data. To join **row1** and **row2**, simply click on **customer\_id** in **row1** and drag it to the expression box of **customer\_id** in **row2**:



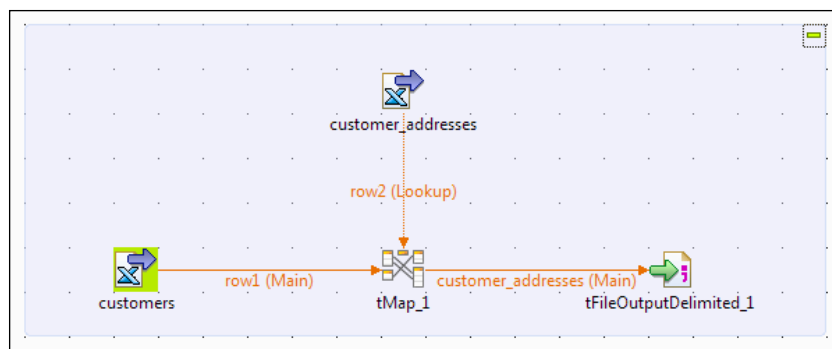
- The component draws a line between the two **customer\_id** fields to show the connection.
- In the output pane of the Map Editor, click on the **+** button to create a new output. Change the name of the output to **customer\_addresses**, to make it more meaningful:



8. Now we can drag-and-drop fields from the input pane to the output pane, building up the mapping and the schema as we go. Multiselect all of the fields from **row1** and drop them onto the **customer\_addresses** output pane. Select **address1**, **city**, **postcode**, and **telephone\_number** from **row2**, and drop these onto the **customer\_addresses** output pane too. Your final mapping configuration should be similar to what is shown in the following screenshot:



9. Now search for a delimited output file in the **Palette** window and add this to the Job Designer. Right-click on the **tMap** component, select **Row | customer\_addresses**, and drop the connector onto the delimited output component.
10. Change the **File Name** field of the delimited output component to something appropriate in the **DataOut** folder. The final job configuration should be as shown in the following screenshot:



It is best practice to layout lookup components, as shown in the previous screenshot. Specifically, the **Main** component is placed on the left-hand side, with one or more **Lookup** components above the main dataflow feeding into it as dataflows from left to right. The positioning does not, of course, change the output of the job in any way, but this is a Talend convention and makes jobs easier to read, particularly in a collaborative development environment, where many developers may work on a job over time.

11. Let's run the job. You should see the data from both the sheets joined into the single delimited file.

Referring back to the original spreadsheet, you can see that there are five distinct customers in the customer sheet, but only four customers in the address sheet. This has been represented correctly in the output file. Take a look at the last line; it has the customers' title, name, and e-mail, but a series of empty fields for the address data:

customer-addresses.csv	
1	12345;Mr;Fitzwilliam;Darcy;fdarcy@darcy.com;1 High Street;Meryton;MY2 3RG;01234-567837
2	12346;Mr;Charles;Bingley;charlesb@bingley.com;22 The Dales;Meryton;MY3 4RJ;01234-678593
3	12347;Ms;Elizabeth;Bennet;ebennet@bennet.com;52 Green Lane;Meryton;MY1 3SH;01234-678294
4	12348;Ms;Charlotte;Lucas;clucas@lucas.com;78 Smith Street;Meryton;MY3 6AR;01234-278303
5	12349;Mr;George;Wickham;gw@wickham.com;;;;
6	

This is because joins created in the **tMap** component are, by default, **left outer joins**.

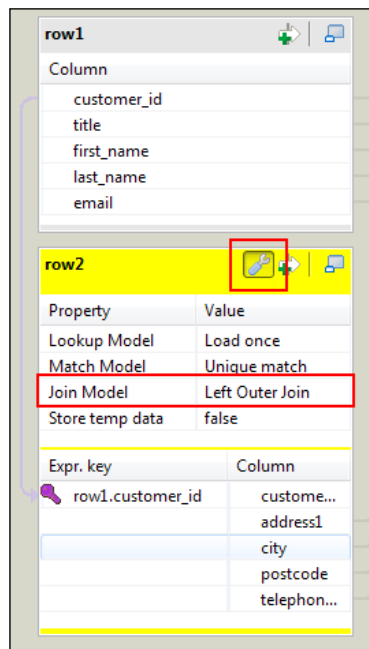


**Left outer joins** is a common concept in database SQL queries, and where there are two database tables, X and Y, which are joined by a query, a left outer join will result in all the records from the left-hand table X being returned, even if there is no matching record in the right-hand table Y.

The result of a left outer join can be clearly seen in the data output from our job.

However, you may not always want the result set to behave in this manner, and you can set the join type, which is a so-called inner join, so that only those records that are represented in both data sets appear in the output.

To configure this in our job, go back to the **tMap** component and click on the spanner icon of **row2**. You'll see some other configuration options revealed:

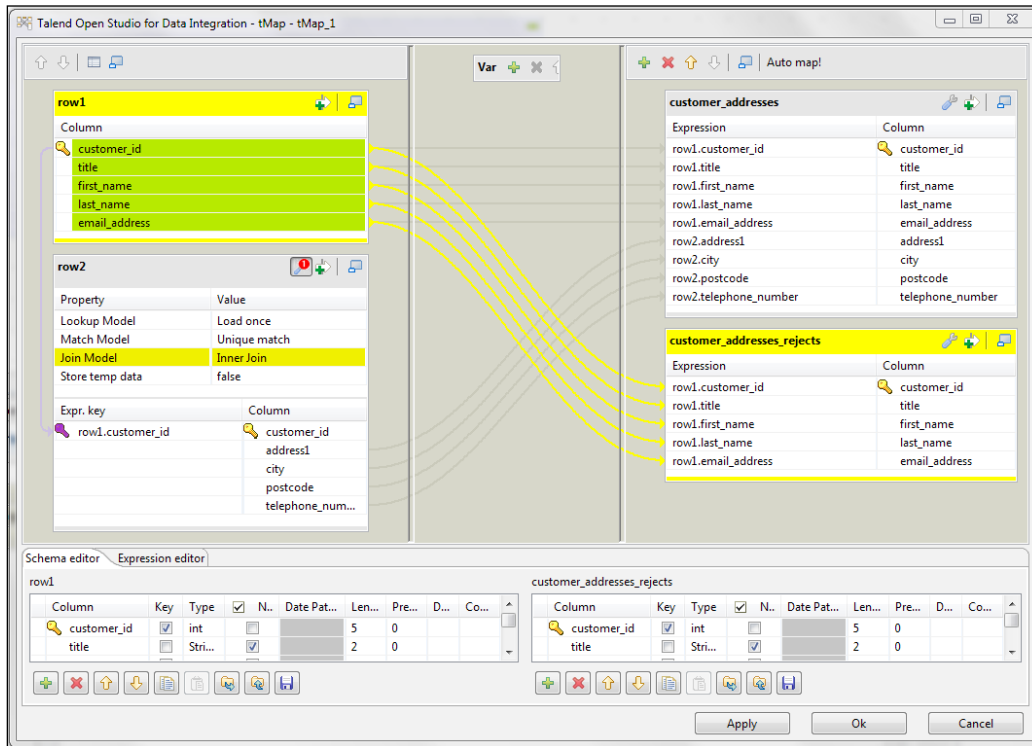


Click on the words **Left Outer Join** and an ellipsis button is revealed. Click on this button, and the pop-up window will show two options for the **Join Model** field, **Left Outer Join** and **Inner Join**. Click on **Inner Join** and then click on **OK**. Click on **OK** to close the **tMap** window and run your job again. This time, you will see that only four rows are returned, only those customer records that have an address.

If you wish to capture the records rejected by the inner join (those that do not have an address), you can define another output in the **tMap** component to achieve this. Double-click on the **tMap** component and perform the following steps:

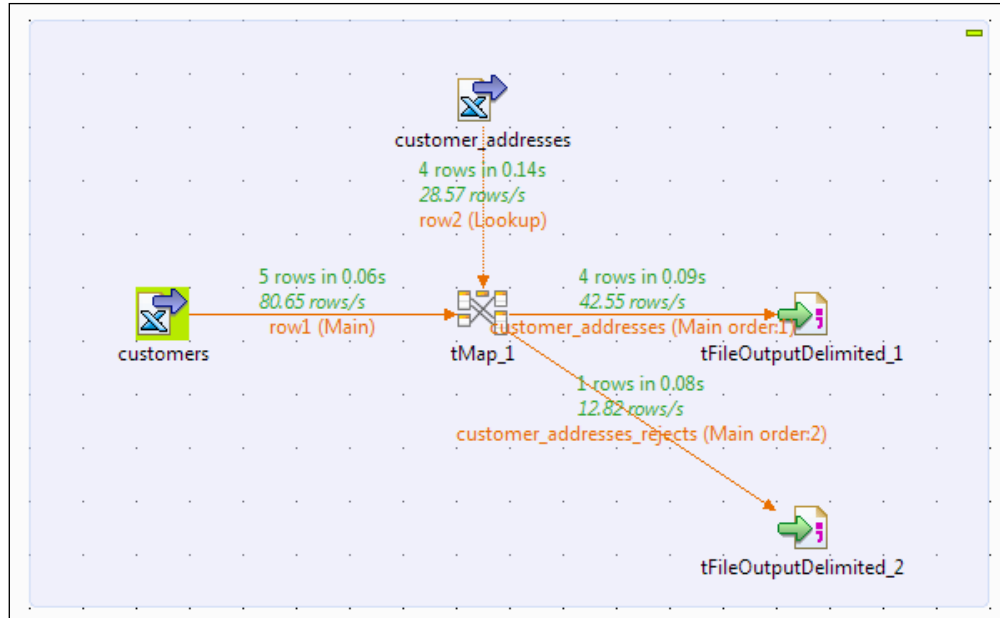
1. Click on the **+** button of the output pane to add a new output. Change its name to **customer\_addresses\_rejects** and click on **OK**.

- Multiselect all of the fields from the **row1** input pane and drag these onto the new output. We don't need any fields from **row2** as we know that any records sent to this new output will not have address data.



- Click on the spanner icon for the **customer\_addresses\_rejects** output pane. This will reveal its settings. Click on the value box of the **Catch lookup inner join rejects** row to reveal the ellipsis button. Click on the button and select **True** from the options. Click on **OK** to accept this change, and then click on **OK** to close the Map Editor and save its settings.
- Add another delimited file from the **Palette** window to the Job Designer. Right-click on the **tMap** component, select **Row | customer\_addresses\_rejects**, and drop the connector onto the new delimited file component.

5. Change the **File Name** parameter of the delimited component and run the job. You will see the four records going to the original delimited output as before, but now the single rejected record will pass to the new delimited component:



## Summary

This was the first time we got our hands dirty with the Studio, the Studio, and we looked at some common integration scenarios based on common file formats. We looked at transforming files from one format to another – CSV to XML, for example. We also explored the **tMap** component and its built-in Expression editor. We built jobs using the multi-schema XML component, and saw how we can map data to the most complex XML structures using the advance XML output component. We also saw how to join data from different sources using both the **tJ join** component and the join functionality within **tMap**.

Let's move on to look at how the Studio can work with another common systems component – the relational database.