



Talend Open Studio: How To Create a Custom Component

Inhoudstafel

1. Introduction	3
2. Requirements	3
3 Location and associated files	4
3.1. JET/template file	5
3.2. Message property file	5
3.3. XML descriptor file	5
4. Technical details	5
4.1. Sections of a component	6
4.2. Initial steps of component creation	7
5. Call to action: your first component!	10
5.1. Adding the XML descriptor file	10
5.2. Adding the message property file	12
5.3. Adding an icon	13
5.4. Adding the JET files	13
5.5. Defining parameters	17
5.6. Parameters in action	20
5.7. Defining connections	21

Talend Open Studio: How To Create a Custom Component

1. Introduction

As you probably already know, Talend is entirely based on a simple concept called “components”. Components can be described as functional pieces to a puzzle, which are graphically represented in the shape of an icon. The puzzle itself is a Talend job, where numerous actions can be performed to your own liking. These actions can be achieved by dragging certain components from the Palette to the job canvas. For example, the tFileInputExcel allows you to read a specific Excel file as well as extract the data contained within that file.

Even though there is a vast amount of components available within Talend, sometimes there might be the need to perform an action that none of these components can offer. That’s where the creation of custom components comes into play. Custom components can be created to fulfill an action based on your own specific needs. The main subject of this tutorial is the process of creating such a custom component from start to finish. Keep in mind that this is just a single possible approach.

The reason why this tutorial was brought to life is the fact that there isn’t that much information available online concerning this subject. The basic tutorial which is provided by Talend in the knowledge base is incredibly short and incoherent. It doesn’t even come close to providing the whole package of information that is required to actually create a custom component. This tutorial sets the record straight. For an even more in-depth explanation of component creation I’d like to refer to [the following link](#). The explanation itself is provided by a BI consulting company called PowerUp.

2. Requirements

In order to follow and complete this tutorial there are a few basic requirements:

- A working distribution of Talend Open Studio. The preferred environment is Talend Open Studio for Data Integration because of its simplicity and straightforward design. Any version above 4.2.3 will do.
- Java knowledge. In order to fully understand this tutorial basic Java knowledge is required. You need to be able to read, write and interpret Java code.

3. Location and associated files

























Whenever Talend is installed on your local file system, the following dedicated directory is present:

<Talend Studio installation dir>/plugins/org.talend.designer.components.local-provider_/components/.

This directory holds all available standard Talend components in separate subfolders. Each subfolder has its own specific name, which corresponds to the name of a component.

If you open one of these subfolders, for example tFileInputDelimited, you'll notice it consists of the following files:

- A few JET (Java Emitter Template) files
- Message property files
- An image file containing the icon associated with the component that is shown in the Palette. This image must have a size of 32*32 and the type has to be PNG.
- An XML descriptor file

	tFileInputDelimited_begin.javajet	10/05/2016 17:10	JAVAJET File	56 KB
	tFileInputDelimited_end.javajet	10/05/2016 17:10	JAVAJET File	5 KB
	tFileInputDelimited_icon32.png	10/05/2016 17:10	PNG File	2 KB
	tFileInputDelimited_java.xml	10/05/2016 17:10	XML Document	7 KB
	tFileInputDelimited_messages.properties	10/05/2016 17:10	PROPERTIES File	2 KB
	tFileInputDelimited_messages_ar.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_de.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_el.properties	10/05/2016 17:10	PROPERTIES File	4 KB
	tFileInputDelimited_messages_en.properties	10/05/2016 17:10	PROPERTIES File	2 KB
	tFileInputDelimited_messages_es.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_fr.properties	10/05/2016 17:10	PROPERTIES File	2 KB
	tFileInputDelimited_messages_hr.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_it.properties	10/05/2016 17:10	PROPERTIES File	2 KB
	tFileInputDelimited_messages_ja.properties	10/05/2016 17:10	PROPERTIES File	2 KB
	tFileInputDelimited_messages_jp.properties	10/05/2016 17:10	PROPERTIES File	2 KB
	tFileInputDelimited_messages_nl.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_pl.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_pt_BR.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_ro.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_ru.properties	10/05/2016 17:10	PROPERTIES File	2 KB
	tFileInputDelimited_messages_slk.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_sr.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_tr.properties	10/05/2016 17:10	PROPERTIES File	1 KB
	tFileInputDelimited_messages_zh_CN.properties	10/05/2016 17:10	PROPERTIES File	1 KB

3.1. JET/template file

These type of files allow you to generate text output based on a certain EMF model. In Talend these files are used to generate Java output code, which is then deployed and compiled inside a job. The file consists of two types of code: template code and Java output code. You can recognize the template code by looking for "<% %>" tags.

3.2. Message property file

In a message property file, you can define the display names or labels for the properties of a certain component. These properties are actually variables that are present in the JET files. Furthermore, this file also contains a description for the component (LONG_NAME). This label is a tooltip of the component. This tooltip is shown whenever you mouse over it in the Palette.

3.3. XML descriptor file

An XML descriptor file, in this case, describes how a component should be deployed. Within the XML structure you can find all the information required to define a component, such as the attributes belonging to the component, configuration requirements, interaction with other components and so on.

4. Technical details

In essence, a component within a Talend job consists of generated Java output code in the form of a snippet. The generated Java output code originates from the template (JET) files. Whenever you drag a component to the job canvas and save the actual job, the Java code is compiled automatically. A job itself is a Java class. Whenever a component is added to a job, the code of the job changes dynamically. Whenever a connection is made with another component or a parameter of a component is modified, the code changes as well. It's all perfectly linked together.

It's very important to note that parameters of a component are only visible to the template. The template basically transforms the parameters into string constants. These constants are then inserted in the Java output code whenever, as well as where they're needed. So whenever you modify parameter values of a certain component, the Java code of a job changes thanks to the template.

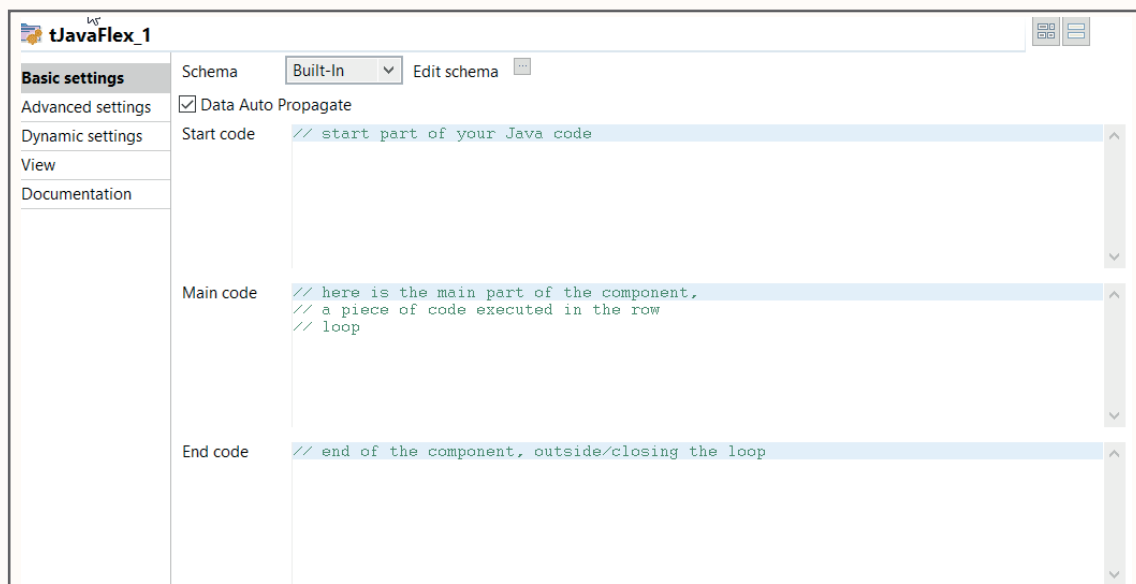
The process that was previously explained can be summarized into the following steps:

- 1 Parameters are defined
- 2 Templates take care of the changes and pass the values
- 3 Java output code changes based on the values that were passed by the templates
- 4 The Talend job consists of the Java output code and is ready to be run with the latest changes

4.1. Sections of a component

A component usually consists of three different sections: begin, main and end. It's perfectly possible to use only one section or two sections but you'll notice that most components have all three sections defined. Each of these sections will generate a separate piece of Java output code, that will be added to the Java output code of a certain Talend job. Remember the JET/template files I've mentioned before? Well, these sections are the actual templates and each template has a dedicated file.

If you've worked with Talend and its components before, then you might be familiar with the concept of these sections already. In this context, one component in particular, namely the tJavaFlex component, might ring a bell. If you look at its basic settings, all of the previously mentioned sections are present.



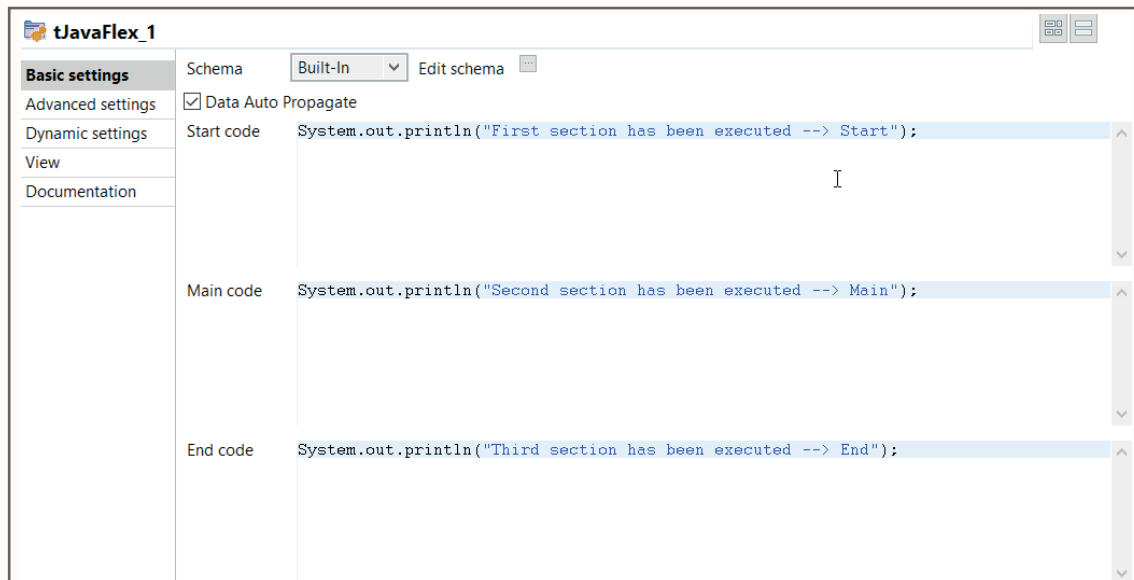
This component however doesn't operate the same way as the templates do within a custom component. You can basically just write code in the sections and it'll be immediately added to the Java output code without any transformations. It's not really hard to figure out that these sections are executed in chronological order. First the start, then the main and ultimately the end section. To be able to see this happen up close, you can create a random Talend job and drag a tJavaFlex component to the job canvas. After that, insert the following

code snippets in the appropriate sections:

```
System.out.println("First section has been executed --> Start");
```

```
System.out.println("Second section has been executed --> Main");
```

```
System.out.println("Third section has been executed --> End");
```



Run the job and, as expected, the following output is printed on the console:

```
[statistics] connecting to socket on port 4047
[statistics] connected
First section has been executed --> Start
Second section has been executed --> Main
Third section has been executed --> End
[statistics] disconnected
```

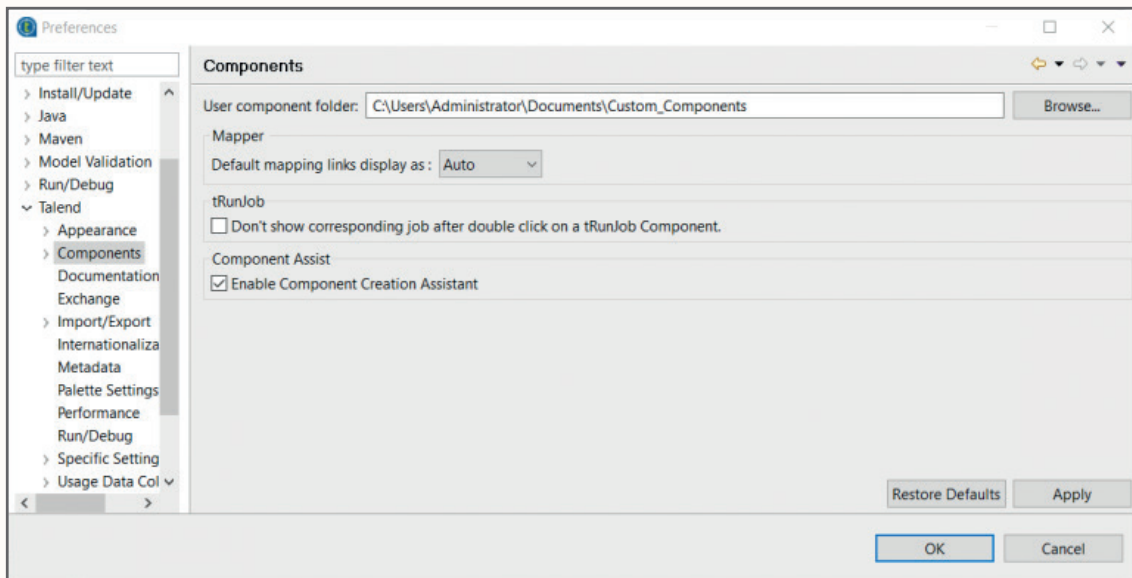
4.2. Initial steps of component creation

There are two possible ways to create your own custom components. You can either do it manually or using a dedicated wizard. Talend Open Studio includes a graphical interface, called the "Component Designer", which is specifically designed for the creation of components. This interface includes the previously mentioned wizard. Personally, I feel like the Component Designer adds a lot of initial junk code to the various files when using the wizard to create a component. For this reason, I prefer to create custom components manually. That is of course a personal choice.

Let's get things started! Before creating an actual component, Talend requires a specific folder to be created somewhere on your local file system. All of the custom components you're going to make, will be stored in this folder. So go ahead and create that folder. I've called mine "Custom_Components".

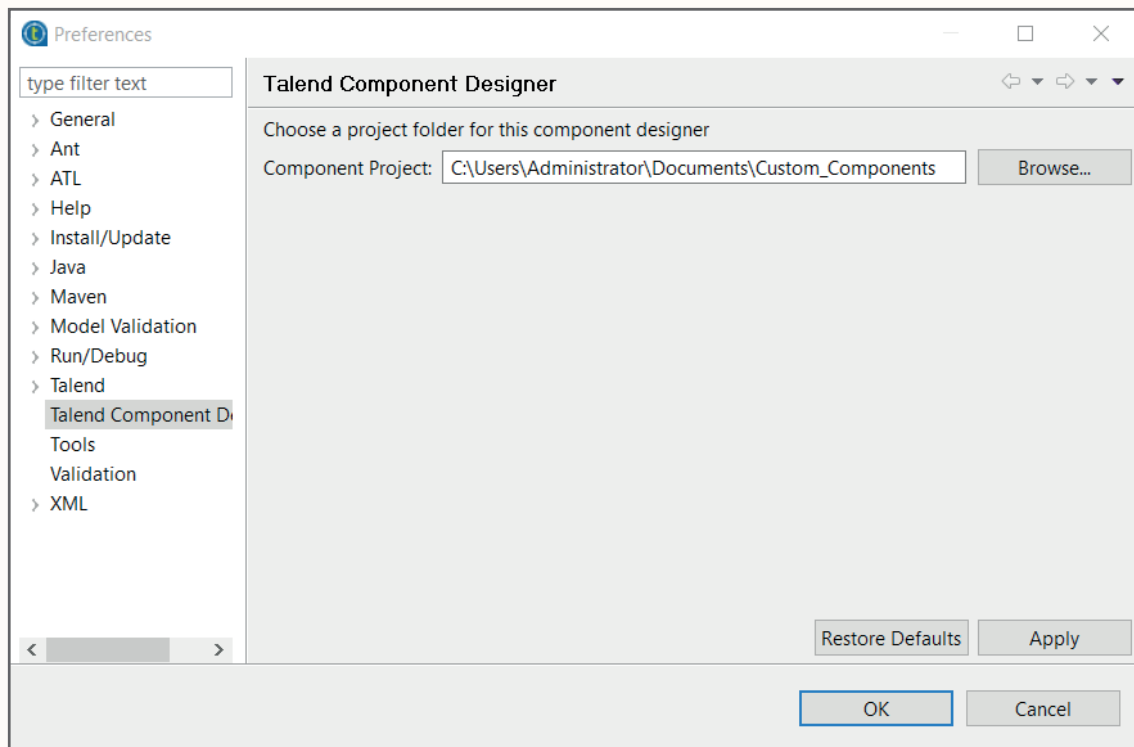
Once the folder has been created, Talend requires its location to be known. In order to do that you have to open your Talend Open Studio and perform the following tasks:

- Go to Windows > Preferences > Talend > Components
- In the blank space next to "User component folder" enter the path to the folder you've just created

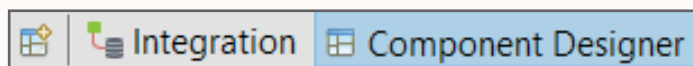


We'll also be using the Component Designer. This interface requires the location of the created folder to be known as well. In order to do that you have to perform the following tasks:

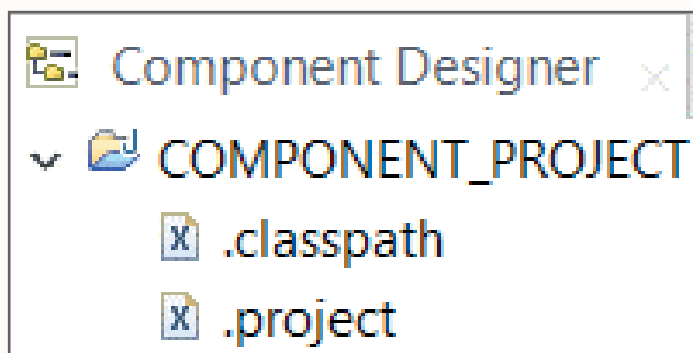
- Go to Windows > Preferences > Talend > Talend Component Designer
- In the blank space next to "Component Project" enter the path to the folder you've just created



Everything is now in place. If you're using Talend Open Studio for Data Integration, there should be two perspectives available in the top right corner. During this tutorial you'll be switching between these two perspectives. Just click on the desired perspective to switch right to it. If, for some reason, one of these perspectives isn't showing up, you can click on the small icon to the left and select the perspective from the list which pops up.



Let's go to the Component Designer perspective. Initially you haven't created a component yet so there aren't any subfolders available. However, you should see the following:



The "COMPONENT_PROJECT" folder is the root folder in the Component Designer perspective. This is a representation of the folder you've previously created somewhere on your file system. In this specific folder you'll create subfolders which correspond to your own custom components.

That's it for now! In the next chapter we'll get to work and create our first component!

5. Call to action: your first component!

In this chapter you'll create your very own first component. We'll start off with a very basic version of a component and then, we'll add some extra functionality to it.

As stated before, I prefer to create custom components manually so throughout this tutorial we'll follow that method. The component that we'll create will be called `tFirstComponent`. First of all, you have to go to your `Custom_Components` folder. Within this folder you have to create a folder with the same name as your component. Go ahead and create that folder. Switch over to Talend and open the Component Designer perspective. If you select the `COMPONENT_PROJECT` root folder and press `F5`, the perspective as a whole is refreshed. The latest changes to the `Custom_Components` folder are then visible. After refreshing the perspective you'll see the folder appear as a subfolder underneath the root folder.

5.1. Adding the XML descriptor file

We'll now start adding files to the newly created `tFirstComponent` folder one by one. These files have already been discussed in the "location and associated files" section. We'll start off with the XML descriptor file. In general, there are only a few parts which are mandatory within the XML structure of this file. There's no information available on this matter. However, thanks to **PowerUp**, a standard structure can be defined. Before defining the actual structure, we have to create the file in the `tFirstComponent` folder. An XML descriptor file has the following naming convention: **NameOfTheComponent_*.java.xml**. Do not use a different convention! It's absolutely imperative to use this one and only this one. It allows Talend to recognize this file as an XML descriptor file after all. Go to your `tFirstComponent` folder and create the `tFirstComponent_*.java.xml` file.

Switch over to Talend again and hit refresh. The XML descriptor file is now visible under the `tFirstComponent` folder. Double-click on the file and it'll open in the editor of the Component Designer perspective. Paste the following structure in the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<COMPONENT>
  <HEADER
    AUTHOR="Jeremy Boterdael"
    COMPATIBILITY="ALL"
    PLATFORM="ALL"
    RELEASE_DATE="20160920"
    SERIAL=""
    STARTABLE="true"
    STATUS="BETA"
    VERSION="0.1">
    <SIGNATURE/>
  </HEADER>
  <FAMILIES>
    <FAMILY>IntoData Tutorial</FAMILY>
  </FAMILIES>
  <DOCUMENTATION>
```

```

        <URL>https://intodata.eu/talend-open-studio-how-to-create-a-custom-
component-part-1/</URL>
    </DOCUMENTATION>
    <CONNECTORS>
        <CONNECTOR CTYPE="FLOW"/>
    </CONNECTORS>
    <PARAMETERS>
    </PARAMETERS>
    <CODEGENERATION/>
    <RETURNS>
    </RETURNS>
</COMPONENT>

```

Let's discuss the different tags which are present in this structure. The first tag we see here is the **HEADER** tag. All of the parameters within this tag are mandatory, however you can give most of them a random value of your choice. It won't affect your component in any way, they just have to be filled in. There is one parameter, called **STARTABLE**, which cannot just get a random value. This parameter does affect the component directly. Whether **STARTABLE** is supposed to be true or false depends on the role of the component you're creating. As you know, some components are classified as input components. These components fetch data from files for example and then pass the data to the next component via a connection. Input components are always located at the start or beginning of a subjob. If you connect the dots, you'll come to a conclusion: these kind of components require the **STARTABLE** parameter to get the value true. A component that receives data, transforms it and then sends it to the next component or a component which can be classified as an output component does not require the value true but the value false.

On a side note, you might notice one parameter is called **PLATFORM**. I have no idea why there's a grammatical error in this word, but you have to write it this way. **PLATFORM** won't work at all so don't try and adjust it.

The next tag is called **FAMILIES**. The inner tag is called **FAMILY** and that's the one I'm going to talk about. Basically, the use of this tag is very easy to explain. When you go to the Integration perspective and create a job there or open an existing job, you'll see the Palette appear on the right side. This specific tag is related to the Palette itself. In general, you'll see the Palette is divided into certain categories or "families" such as Big Data, Business Intelligence, Business, Cloud, ... The **FAMILY** tag allows you to specify your very own category or "family". Later on, we'll push our component to the Palette and your own specified family will then become visible.

There's also a **DOCUMENTATION** tag present. The use of this tag should be quite clear. It allows you to add documentation or useful information by specifying a URL between the **URL** tags. As a point of reference I've added a link to the first part of this tutorial.

Next up is the **CONNECTOR** tag. This tag allows you to specify the inputs and outputs of your component. It indicates how components interact with each other depending on their role. As you can see, we specified a **CTYPE** parameter and given it the value "FLOW". This is the most common connector type. It matches with the usual "row<number>" connections between components. Additional information can of course be added to this tag, but we'll get into that later.

The **PARAMETERS** tag allows you to specify input parameters for your component. These parameters are used to pass information to the component itself based on the needs of the user. In other words, they're the options most of the standard Talend components provide. You can choose which parameters apply to your own component. As an example you can take a look at the parameters of the tFileInputDelimited component:

The **CODEGENERATION** tag will not be discussed because it won't be part of this tutorial. The final tag is called **RETURNS**. The inner tag, called **RETURN**, allows you to specify the output of the component, which will be used afterwards. The most commonly used return parameter is a global variable called `NB_LINE`. This returns the amount of records that has been handled by the component. I've provided an example of the usage below.

```
<RETURN NAME="NB_LINE" TYPE="id_Integer" AVAILABILITY="AFTER"/>
```

5.2. Adding the message property file

The XML descriptor file has been completely discussed. Now, let me explain a very important step in the process of creating a component. If you switch over to the Integration perspective, you'll notice that your component isn't available yet in the Palette. In order for that to happen you have to push your component to the Palette. To push the component to the Palette you have to right-click the subfolder, which is present in the Component Designer perspective called `tFirstComponent`, and then click on "Push Components to Palette". Remember to always hit refresh whenever you add a file to the subfolder before you push your component to the Palette. Right now pushing your component to the Palette won't do anything at all. It'll say that the component has been published but it won't be visible just yet. The reason for that is the fact that you need another file to actually make it appear. That file is the next one in line: the message property file.

This file is absolutely mandatory. Go ahead and create another file in the `tFirstComponent` subfolder called `tFirstComponent_messages.properties`. After that switch over to Talend and open the Component Designer perspective. The first thing you have to do is refresh the `tFirstComponent` subfolder. You'll see that the newly created file's been added. Now open the file and paste the following text in it:

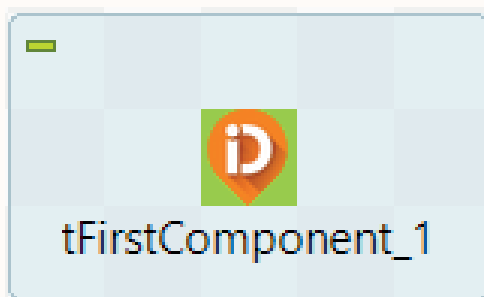
```
LONG_NAME=My very own first component
```

You can add comments in this file using the `"#"` mark. What this line provides is a label for the **LONG_NAME** parameter of the component. It's that simple. Later on we'll see an example of this sort of labeling. This is just a standard line, which is always present in this file.

It's now possible for you to push the component to the Palette so go ahead and do that. If you haven't created an empty job yet, then please do it right now. Switch over to the Integration perspective. If you've followed this tutorial very carefully and you've pushed your component to the Palette, then you should see the family you've previously defined appear in the Palette. For me that would be "IntoData Tutorial", for you it might be something else. If you click on this family, you should see your component in the shape of an element belonging to it.

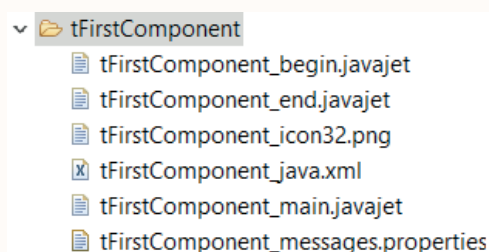
5.3. Adding an icon

So there you have it! You can see your component in the Palette! However, do not drag your component to the project canvas and use it. I cannot stress this enough. If you would do that, a null pointer exception will pop up and ruin your day. Why exactly is this null pointer exception taking place? Well, because we're missing another element: an icon for the component. Remember, you need an icon with a size of 32*32 and it has to be in a PNG format. Go ahead and make an icon or download one, whichever you prefer. Once you've done that, you have to rename the icon to `tFirstProject_icon32.png` and put it in the `tFirstComponent` folder. After you've done that, all you have to do is hit refresh and push the component to the Palette once more. Switch over to Integration perspective, check your component again and you'll notice that the icon has changed. Drag the component to the job canvas and then run the job. Of course, as stated before, nothing happens but there's no exception either. As a reference, my component looks like this when dragged to the canvas:



5.4. Adding the JET files

The basic required files are all set up. The component is visible in the Palette and it can be used in a job. I'd say it's about time to write some code! If you've followed this tutorial from beginning to end so far, you'll notice a set of important files hasn't been discussed yet: the JET files. Remember that I've told you that a component consists of sections, namely the begin, main and end section. Each section is represented by a JET file. Let's start creating these files. What I want you to do is: create three files and call them `tFirstComponent_begin.javajet`, `tFirstComponent_main.javajet` and `tFirstComponent_end.javajet`. Switch over to Talend, open the Component Designer perspective and hit refresh. The final setup of your folder should look like this:



Okay, let's proceed. Open the `tFirstComponent_begin.javajet` file. In each of the JET files classes are imported at the beginning of the file. These classes vary and are entirely based on your needs. If you need a certain class, you just import it there. In this tutorial we're going to use some basic import statements. These statements are present when you use the Component Designer wizard to create these files so we can state that they're default. Go ahead and paste the code below in the file:

```

<%@ jet
imports="
org.talend.core.model.process.INode
org.talend.core.model.process.ElementParameterParser
org.talend.core.model.metadata.IMetadataTable
org.talend.core.model.metadata.IMetadataColumn
org.talend.core.model.process.IConnection
org.talend.core.model.process.IConnectionCategory
org.talend.designer.codegen.config.CodeGeneratorArgument
org.talend.core.model.metadata.types.JavaTypesManager
org.talend.core.model.metadata.types.JavaType
"
%>

```

Please note that you'll need these import statements in every JET file. Of course it's just a reference and they will always vary in each file separately depending on your needs. So go ahead and open the `tFirstComponent_main.javajet` and `tFirstComponent_end.javajet` as well and paste the above code in those files.

The example I've given at the beginning of the tutorial, to show you guys how the sections work using the `tJavaFlex` component, will be used here again. I'd like you to open the `tFirstComponent_begin.javajet` file, if it isn't open already, and paste the following code below the import statements (after the closing tag):

```

System.out.println("First section has been executed --> Start");

```

This is just general java output code. The point is to print something on the console when the component is used in a certain job and that job gets executed. Hit refresh and then push the component to the Palette. Switch over to the Integration perspective. If you've already created a job, you can drag the component to the job canvas and execute the job. If the component is on the canvas already then you'll get a pop up saying the component has to be reloaded when executing the job. You should be able to see the above message appear on the console.

I'd like to note that I've had quite a few problems in the beginning with this exact step. For some reason the message was never displayed on the console, meaning that the changes weren't pushed to the Palette for some reason. Possible solutions for this are:

- If the component was already present in a certain job, delete the component and then drag it back to the job canvas. After that execute the job again and it should work.
- If the above doesn't work, then you have to add an extra step to the solution. Basically all you have to do is go back to the Component Designer first, hit refresh again and push the component to the Palette. Then follow the above steps.
- If these steps don't solve your problem either, you can add one more step to the process. Just restart Talend first and then follow all of the previously described steps.

To complete the example, I'd like you to open the `tFirstComponent_main.javajet` and paste the following code below the import statements:

```
System.out.println("Second section has been executed --> Main");
```

After that open the tFirstComponent_end.javajet and paste the following code below the import statements:

```
System.out.println("Third section has been executed --> End");
```

Then follow the usual two steps of refreshing and pushing the component to the Palette. Go to your job and execute it. The output should be as follows:

```
Starting job IntoDataComponentTutorial at 13:18 13/09/2016.
[statistics] connecting to socket on port 3705
[statistics] connected
First section has been executed --> Start
Second section has been executed --> Main
Third section has been executed --> End
[statistics] disconnected
Job IntoDataComponentTutorial ended at 13:18 13/09/2016. [exit code=0]
```

I'd like to expand the example with a loop to prove a certain point. In this loop we'll just run a certain text 5 times. I'd like you to go to the tFirstComponent_begin.javajet file and paste the following code below the previously written code:

```
for (int counter=1;counter<=5;counter++)
{
```

Open the tFirstComponent_main.javajet file and change the previously written code to:

```
System.out.println("The main section has been executed " + counter + " time(s)");
```

Open the tFirstComponent_end.javajet file and add a closing tag ("}") there:

```
}
System.out.println("Third section has been executed --> End");
```

Perform the usual steps in the Component Designer, switch over to the Integration perspective and execute your job. You should get the following output:

```
Starting job IntoDataComponentTutorial at 14:01 13/09/2016.
[statistics] connecting to socket on port 3371
[statistics] connected
First section has been executed --> Start
The main section has been executed 1 time(s)
The main section has been executed 2 time(s)
The main section has been executed 3 time(s)
The main section has been executed 4 time(s)
The main section has been executed 5 time(s)
Third section has been executed --> End
[statistics] disconnected
Job IntoDataComponentTutorial ended at 14:01 13/09/2016. [exit code=0]
```

The reason why I've asked you to add this loop was to point out something very interesting. I hope you noticed we opened the loop in the begin section, then added the output code in

the main section and finally closed the loop in the end section. The point I'm trying to make here is that these sections are executed as one big block, one after another. Behind the scenes code snippets are joined together in that block. This allows for a certain amount of flexibility when writing code in these files.

Let's take a look at an interesting feature provided by Talend and change the code once again. When you use a certain component multiple times in the same job, there has to be a way to differentiate which variables in the code belong to which component. If that wouldn't be the case, you'd get errors all over your Talend job stating that variables are the same in multiple elements and thus creating contradictions. The solution to this problem is UNIQUE NAME. This element adds a unique name to variables. This unique name is given to an instance of a component. So, if you have two of the same components in the same job these components will each have a different unique name. So always add the unique name when creating a local variable. I'll show you exactly how to do that. I can already give away that this is really easy.

First of all, open the tFirstComponent_begin.javajet file and add the following piece of template code between the import statements and the written code below it:

—import statements—

```
<%  
CodeGeneratorArgument codeGenArgument = (CodeGeneratorArgument) argument;  
INode node = (INode)codeGenArgument.getArgument();  
String cid = node.getUniqueName();  
%>
```

—previously written code—

When you use the Component Designer wizard, this piece of template code is added to every JET file automatically. Of course we've decided to do this manually so you have to add it yourself. I'm not going to get into too much detail. What you need to know is that an instance in a job is called a node. With the getUniqueName() method you can retrieve the unique name or the instance of that specific node. The unique name is then stored in the variable called cid (Component ID).

Let's adjust the code we've previously written too. Change the code to the following:

```
System.out.println("First section has been executed --> Start");  
for (int counter_<%=cid %>=1;counter_<%=cid %>=<=5;counter_<%=cid %>++)  
{
```

With the <%= %> we're passing the values from the template code to the java output code. Now go to the tFirstComponent_main.javajet file and add the above template code to it. Then change the previously written code to the following:

```
System.out.println("The main section has been executed " + counter_<%=cid %> + "  
time(s)");
```

Follow the usual steps in the Component Designer then go ahead and execute your job again. Everything should still work. I'd like you to take a look at the code of the job itself by selecting the "Code" tab, which is right next to the "Designer" tab in the job window. Look for the defined variable called counter. You'll notice that the addition of "_<%=cid %>" in the code has led to the transformation of the cid variable into a string representing the unique name of the component in the java output code of the job. Use this best practice and necessity at all times!

5.5. Defining parameters

As stated before, you can add parameters to your custom component. Parameters are used to define the settings of a component. There are two types of parameters: basic parameters and advanced parameters. In essence they're the same, however basic parameters imply necessary settings or key settings and advanced parameters imply optional settings. Of course, it's not mandatory to define parameters, it's just a possibility. Once you've defined these parameters, you can see them in the "Basic settings" and "Advanced settings" panel of the component.

Where exactly are these parameters defined? Well, if you've followed the tutorial all the way up to this point, you might remember that there's a **PARAMETERS** tag in the XML descriptor file. Currently that tag doesn't contain anything yet, which means no parameters are defined. This specific tag refers to the basic parameters of a component. If you don't want to define any basic parameters, you can just specify the following: `<PARAMETERS/>`. Next to this tag, there's a separate tag for the advanced parameters of a component called **ADVANCED_PARAMETERS**. If you don't want to define any advanced parameters, you can just specify `<ADVANCED_PARAMETERS/>` or leave this tag out of the XML descriptor file in general. It's already been proven that you can actually leave the tag out, because we've never even added it to our current XML descriptor file and the component works perfectly.

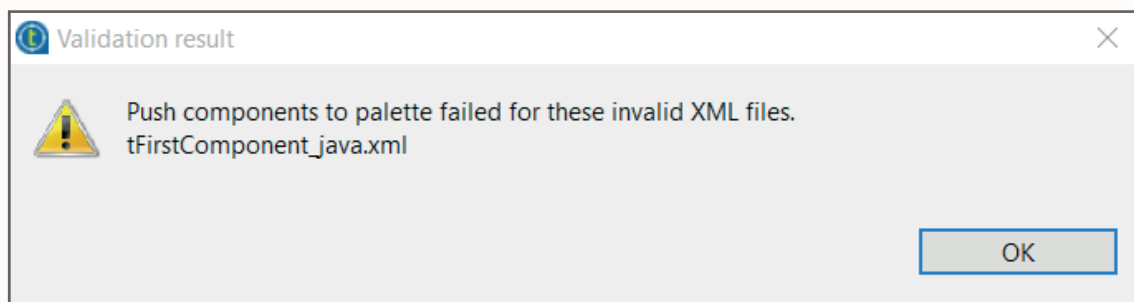
I'd like you to open the `tFirstComponent_java.xml` file, the XML descriptor file, and paste the following code between the `CONNECTORS` tag and the `CODEGENERATION` tag:

```
<PARAMETERS>
  <PARAMETER
    NAME="FirstBasicParameter"
    FIELD="TEXT"
    REQUIRED="true"
    NUM_ROW="1" >
    <DEFAULT>"Insert text"</DEFAULT>
  </PARAMETER >
  <PARAMETER
    NAME="SecondBasicParameter"
    FIELD="CHECK"
    REQUIRED="false"
    NUM_ROW="2" >
    <DEFAULT>>false</DEFAULT>
  </PARAMETER >
</PARAMETERS>
<ADVANCED_PARAMETERS>
  <PARAMETER
    NAME="FirstAdvancedParameter"
    FIELD="INTEGER"
    REQUIRED="true"
    NUM_ROW="1">
    <DEFAULT>0</DEFAULT>
  </PARAMETER>
  <PARAMETER
    NAME="SecondAdvancedParameter"
    FIELD="MEMO_SQL"
    REQUIRED="false"
    NUM_ROW="2">
    <DEFAULT>"select * from employee"</DEFAULT>
  </PARAMETER>
</ADVANCED_PARAMETERS>
```

As you can see we added a total of 4 parameters, 2 basic parameters and 2 advanced parameters. Let's take a closer look at the attributes which are defined for every parameter:

- **NAME:** With this attribute you define the unique name of a component. This name will be used to reference the parameter in the code of the JET files.
- **FIELD:** With this attribute you indicate what the parameter can store. The default value is "TEXT", meaning that the parameter can store a string. There are multiple possibilities though. When you look at the parameter "SecondBasicParameter", the value "CHECK" has been given to the attribute. This means that the parameter can verify whether a checkbox has been checked or not. In the background a boolean value is being interpreted, which is either true or false. The content of the parameter, e.g. a text box or a checkbox, is also determined by the value of this attribute.
- **REQUIRED:** This attribute indicates if a job will actually start or not, depending whether or not a value was given to the parameter. This implies that if this attribute gets the value false, a job will be run even if the parameter wasn't given a value. If this is not the case and it gets the value true, then the job will not be run if the parameter wasn't given a value.
- **NUM_ROW:** This attribute specifies where a certain parameter will appear in the "Basic settings" or "Advanced settings" panel, depending on the type of parameter. It represents the relative line number. For example, the "FirstBasicParameter" parameter will appear on the first line and the "SecondBasicParameter" parameter will appear on the second line, or in other words underneath, in the "Basic settings" panel. Of course, it's perfectly possible to have multiple parameters on the same line directly next to each other. In that case, all you have to do is give those parameters the value 1.

Both the NAME and FIELD attributes are mandatory attributes when adding a parameter to a component. If you don't use these attributes but only the REQUIRED attribute for example, the following error will be returned:



So make sure these attributes are present. You'll notice that, inside the parameter tag, there's another element called DEFAULT. This element allows you to set a default value for the parameter. When a parameter requires you to insert a certain string, you can set a default value of "hello world" for example.

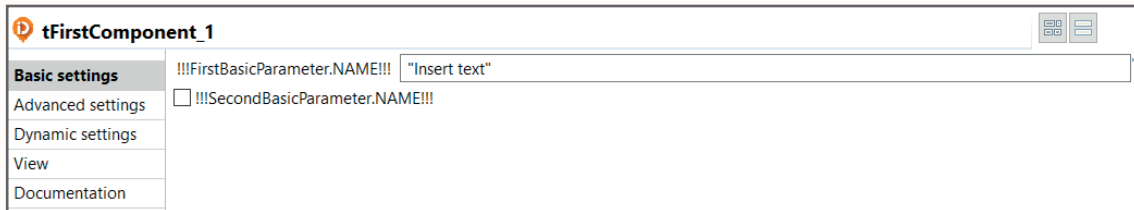
We've now handled just a few possible attributes, however there are more attributes available. To get a list of these, you can go to the following link:

<http://www.talendbyexample.com/talend-custom-component-xsd.html>

The parameters are all set. Go ahead and push the component to the Palette. Open the job you've made for this tutorial and, if the custom component is on the job canvas, delete it and drag it over there again. I do this every single time a change is made to the component concerning parameters, to make sure the latest changes are visible. You can also just execute the job again. A pop up will appear stating that the component has to be reloaded. Just

click “OK” and the job will be executed with the updated component.

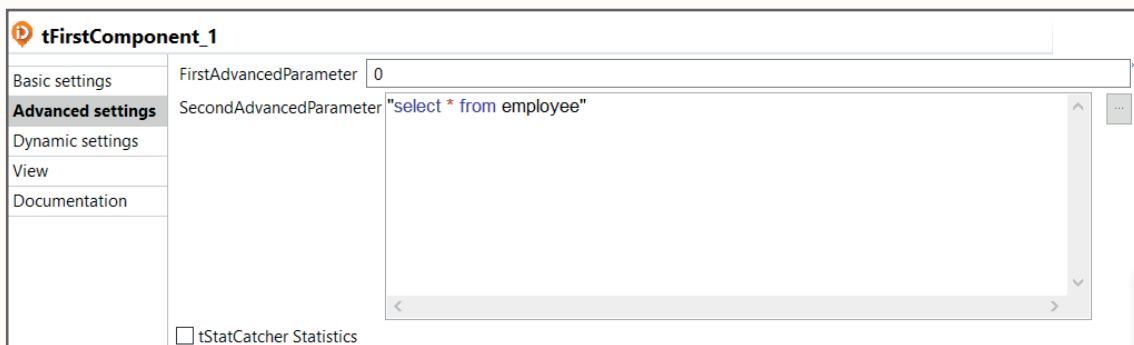
Once the component is updated, take a look at the “Basic settings” panel and the “Advanced settings” panel. I hope you noticed that the labels of the parameters contain very weird names, like “!!!FirstBasicParameter.NAME!!!”, as seen in the image below:



The reason for that is the fact that we haven’t specified a label for the parameters yet. The weird names are the default labels. Let’s go ahead and specify these labels. Despite what you might think, these labels are not defined in the XML descriptor file but in the message property file. Open the `tFirstComponent_messages.properties` file and add the following lines:

```
FirstBasicParameter.NAME=FirstBasicParameter  
SecondBasicParameter.NAME=SecondBasicParameter  
FirstAdvancedParameter.NAME=FirstAdvancedParameter  
SecondAdvancedParameter.NAME=SecondAdvancedParameter
```

Now push the component to the Palette again and follow the previously described steps. You should see the following in each tab:



5.6. Parameters in action

The parameters have been defined and are equally visible when checking the component. Now it’s time to put these parameters to use! As stated before, you can reference a certain parameter in the JET files. Basically, the point is to get the value that a parameter contains. There’s a class called **ElementParameterParser** which allows us to do exactly that. It gets

the value from a node, which is an actual parameter, with a method called **getValue**. The specification of this method is the following: `ElementParameterParser.getValue(IElement node, java.lang.String text)`. It's important to note that this method will always return a `String`, regardless of the data type you might've defined when creating your parameters in the XML descriptor file. This means that all parameters are of the type `String`.

Let's add some more code to the `tFirstComponent_begin.javajet` file (the code which isn't present yet, is marked in bold):

—import statements—

```
<%
CodeGeneratorArgument codeGenArgument = (CodeGeneratorArgument) argument;
INode node = (INode)codeGenArgument.getArgument();
String cid = node.getUniqueName();
String basicTextBox = ElementParameterParser.getValue(node, "__FirstBasicParameter__");
String basicCheckbox = ElementParameterParser.getValue(node, "__SecondBasicParameter__");
int numberOfIterations = Integer.parseInt(ElementParameterParser.getValue(node, "__FirstAdvancedParameter__"));
%>
```

—previously written code—

First of all, make sure that the `"org.talend.core.model.process.ElementParameterParser"` import statement is present in the imports section. It imports the `ElementParameterParser` class, which is necessary to be able to use it in the code. Now take a look at the `getValue` method. For each parameter a separate variable was created. Those variables then get the value that is returned from the method. The method itself has 2 parameters: a node parameter, `node` being an instance of `INode`, and a string parameter representing the unique name of the parameter which was defined in the XML descriptor file. I hope you noticed the second parameter requires a prefix and a suffix, namely `"__"`, which is a double underscore. Please keep in mind that you have to add these statements to the JET template, so between `"<% %>"` tags. In this piece of code, the first and second basic parameters are used, as well as the first advanced parameter. The second advanced parameter isn't used, because in the remainder of the code we won't do anything with it. You can of course add some code for this parameter later on if you want.

In the remainder of the code we're going to add some functionality to the parameters. For the first basic parameter, we want the text, which is inserted in the text box to be printed on the console. This means that the parameter will be used to generate Java output code. For the second basic parameter, we want to print a certain text/string on the console whenever the checkbox is checked. In this case the parameter will be passed as a constant to the Java output code. For the first advanced parameter, we want the number, which is inserted in the text box, to represent the amount of iterations that the previously defined loop goes through.

Go ahead and adjust the code in the JET file to the following (yet again, code changes are marked in bold):

```
<%
CodeGeneratorArgument codeGenArgument = (CodeGeneratorArgument) argument;
INode node = (INode)codeGenArgument.getArgument();
String cid = node.getUniqueName();
```

```
String basicText = ElementParameterParser.getValue(node, "__FirstBasicParameter__");
String basicCheckbox = ElementParameterParser.getValue(node, "__SecondBasicParameter__");
int numberOfIterations = Integer.parseInt(ElementParameterParser.getValue(node, "__FirstAdvancedParameter__"));
%>
```

```
System.out.println("<%=basicTextBox %>");
```

```
<%
if (basicCheckbox.equals("true"))
{
%>
```

```
System.out.println("Checkbox in the Basic settings tab is checked!");
```

```
<%
}
%>
```

```
System.out.println("First section has been executed --> Start");
for (int counter_<%=cid %>=1;counter_<%=cid %> <= <%=numberOfIterations %>;counter_<%=cid %>++)
{
```

The code is self-explanatory. However, take a look at the if statement. You'll notice that we're opening and closing the "%" tags. We're doing that to enter and exit the template when needed. Remember this when writing future code in these files. Go ahead and push the component to the Palette again. Go to the Integration perspective and open your job again. Delete the component, drag it on the canvas and run the job. The output is entirely based on your configurations. By changing the parameter values you'll notice the output changes time and time again.

5.7. Defining connections

There's one more important concept left to handle, namely connections. In general, components interact with each other via a connection, specifically a row connection. I'm sure you're familiar with the different types, such as "Main", "Reject", "Output", "Lookup", ... When defining such connections for your custom component, you'll have to use a connector. We're mainly going to focus on a specific type of connector, called a flow connector. This connector allows data to flow in and/or out of a component.

When it comes to data processing, components are divided in three different categories:

- **Input components:** These components only have outgoing connections.
- **Output components:** These components only have incoming connections.
- **Processing components:** These components have both incoming and outgoing connections.

Usually you already know which category applies to your component. Based on that fact, you can easily define the necessary settings for it.

Let's take a look at our XML descriptor file. You'll notice that a connector is already present, namely within the **CONNECTORS** tag:

```
<CONNECTORS>
<CONNECTOR CTYPE="FLOW"/>
</CONNECTORS>
```

Connectors are defined in an inner tag called **CONNECTOR**. For this specific tag you can define attributes, one being of the utmost importance when it comes to defining a connector, namely **CTYPE**. This attribute is used to specify the type of connector. Of course there are different types of connectors available and it's possible to define multiple connectors in general. However, there should only be one connector per type. In our case the value "FLOW" is set, to indicate that the connector has to be a flow connector. This is exactly what we wanted in the first place. For each type of connector, you can define optional attributes. The most important optional attributes are the following:

- **MIN_INPUT**: This attribute allows you to specify the minimum number of permitted input instances for the connector.
- **MAX_INPUT**: This attribute allows you to specify the maximum number of permitted input instances for the connector.
- **MIN_OUTPUT**: This attribute allows you to specify the minimum number of permitted output instances for the connector.
- **MAX_OUTPUT**: This attribute allows you to specify the maximum number of permitted output instances for the connector.

Using these attributes, the minimum and maximum incoming and outgoing connections can be defined for the component. As stated before, these are optional. However, in the case of a flow connector, maximum input instances and maximum output instances are usually specified. Let's go ahead and specify these attributes for our custom component. We're going to make the component an input component, so only outgoing connections apply. Head over to the XML descriptor file and update the connector statement to the following:

```
<CONNECTOR CTYPE="FLOW" MAX_INPUT="0" MAX_OUTPUT="1"/>
```

Obviously there are no input instances and there's just one output instance. This is only the first step in defining an input component though. Basically there's something very important missing. If you look at the component now, we haven't defined any input data yet nor have we defined a data structure for that input data. That means the component doesn't know which type of data should be sent to the output. In order for this to happen, we have to define a parameter containing the schema/metadata for the output. A parameter is, as you already know, defined in the XML descriptor file. Add the following code to the file:

```
<PARAMETER
  NAME="SCHEMA"
  FIELD="SCHEMA_TYPE"
  REQUIRED="true"
  NUM_ROW="1">
</PARAMETER >
```

The field **SCHEMA_TYPE** takes care of the metadata. Whenever you define a schema, it'll be automatically passed to the outgoing connections. This parameter will be added to the "Basic settings" panel. In the previous part we've already defined two other basic parameters,

so I'd like you to adjust the NUM_ROW attributes for those parameters as well. You basically want three parameters under each other in that specific panel.

Now we're going to write some code, which will allow us to send data to the output. As you already know, we've got three JET files. In the tFirstComponent_begin.javajet file we've got a loop defined. The content of the loop, which generates the actual data output, is defined in the tFirstComponent_main.javajet file. Each time the code in the tFirstComponent_end.javajet file is called, data's being sent to the output connection(s). Per outgoing connection, we want to load the values for every single field contained in each record, exactly before that code is called. These fields are defined in the schema. So our next step involves writing some more code in the tFirstComponent_main.javajet file. Go to that file and start off by adding **java.util.List** to the imports section. I'd like to note that this file is going to change quite a bit. I recommend to override the existing code, underneath the imports section, with the next couple of code fragments. Let's start by adding this piece of code (changes are in bold):

```
<%  
CodeGeneratorArgument codeGenArgument = (CodeGeneratorArgument) argument;  
INode node = (INode)codeGenArgument.getArgument();  
String cid = node.getUniqueName();  
  
List<IMetadataTable> metadatas = node.getMetadataList();  
if ((metadatas != null) && (metadatas.size() > 0)) {  
IMetadataTable metadata = metadatas.get(0);  
if (metadata != null) {
```

In this piece of code, we get the metadata of the component, which is defined in the parameter we just added in the XML descriptor file. You'll notice that a list of MetadataTables is present in the code. In general, it's possible to have multiple SCHEMA_TYPE parameters, each with their own schema. In order to keep track of all of these parameters and schemas we have to put of all them in a list. Of course, in this case, there's only one parameter. To get that specific parameter's schema, we use the **metadatas.get(0)** statement.

For the next part, please add the following statement to the imports section: **org.talend.core.model.process.EConnectionType**. After that, paste the following code below the previously added code:

```
List<? extends IConnection> outgoingConns = node.getOutgoingConnections();  
for (IConnection conn : outgoingConns)  
{  
if (conn.getLineStyle().equals(EConnectionType.FLOW_MAIN))  
{  
String outputConnName = conn.getName();
```

In this piece of code, we put all of the outgoing connections in a list. After that we loop through them in order to get the name of the outgoing **row<number>** connection. The name of the connection is then stored in a String variable called outputConnName. In the loop there's an if statement as well. This statement makes sure only the "MAIN" data type connections are checked. In this case, it's the only connection which will be used so it makes sense to only check that particular one.

Let's add the last piece of code:

```
for( IMetadataColumn col : metadata.getListColumns() ) {  
%>
```

```

<%=outputConnName %>.<%=col.getLabel() %> = "Row data for the <%=col.getLabel()
%> column has been passed!";
<%
}
}
}
}
}
%>
System.out.println("The main section has been executed " + counter_<%=cid %> + "
time(s)");

```

This is quite a complicated structure and it's incredibly disordered when looking at it in different pieces. For that reason, you'll find an image below representing the whole structure:

```

List<IMetadataTable> metadatas = node.getMetadataList();
if ((metadatas != null) && (metadatas.size() > 0)) {
    IMetadataTable metadata = metadatas.get(0);
    if (metadata != null) {
        List<? extends IConnection> outgoingConns = node.getOutgoingConnections();
        for (IConnection conn : outgoingConns)
        {
            if (conn.getLineStyle().equals(EConnectionType.FLOW_MAIN))
            {
                String outputConnName = conn.getName();
                for (IMetadataColumn col : metadata.getListColumns() ) {
                    <%=outputConnName %>.<%=col.getLabel() %> = "Row data for the <%=col.getLabel() %> column has been passed!";
                }
            }
        }
    }
}
System.out.println("The main section has been executed " + counter_<%=cid %> + " time(s)");

```

This last bit of code iterates over the columns defined in the **metadata** variable and writes output to the Java output code. The string inside the for loop, containing the name of a column of an input row, is assigned to an output row. The assignment is done column by column. I've chosen this particular string purely as an example. It's important to note here that I assume that every single column that's defined in the schema is a string. When printing the results to the console, you'll notice that the same string is repeatedly printed for every column. The amount of printed lines depends entirely on the amount of processed rows. For example, if you've got two rows of input data you'll see that the string is printed for each column within a particular row. Two rows containing data means two lines of output. As an extra, I'm printing the amount of times the main section gets executed. Each time it gets executed, a row is processed. This gives you a nice overview in the console window.

In order to test the component, you have to perform the usual steps of pushing the component to the Palette and then dragging it to the job canvas. On top of that, since the component is now an input component, you'll have to drag a tLogRow component to the canvas as well. Finally, connect the custom component to the tLogRow component. Once that's done, you have to define a schema for the custom component. To do that, go to its "Basic settings" and click on "Edit schema". You can choose the schema yourself, however all columns/fields have to be strings! Below you can find my example:

tFirstComponent_1								
Column	K...	Type	<input checked="" type="checkbox"/> N..	Date Pattern (Ctrl+S...	Length	Precision	Default	Comment
End	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>					
Of	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>					
Tutorial	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>					

After that, go to the “Advanced settings”. Remember the first advanced parameter we defined in the previous part of this tutorial? Well, you have to give that parameter a value as well. In this particular context, the value you give to this parameter represents/simulates a certain amount of rows. So, if you give it the value 2, two fictive rows will be generated. In the background all you do is loop through the process twice. Let’s go ahead and give it that value. Now run the job! You should get the following output:

```
Starting job IntoDataComponentTutorial at 14:20 27/09/2016.
[statistics] connecting to socket on port 3586
[statistics] connected
Insert text
First section has been executed --> Start
The main section has been executed 1 time(s)
Row data for the End column has been passed||Row data for the Of column has been passed||Row data for the Tutorial column has been passed!
The main section has been executed 2 time(s)
Row data for the End column has been passed||Row data for the Of column has been passed||Row data for the Tutorial column has been passed!
Third section has been executed --> End
[statistics] disconnected
Job IntoDataComponentTutorial ended at 14:20 27/09/2016. [exit code=0]
```

And here you go, the basics of custom component creation have been handled. I'd like to emphasize that this tutorial is just a start, something to get you going. There's way more to it than this. The best way to learn more about this topic is to pick a random standard Talend component and check out the code in the different files belonging to that component.

Used sources:

<http://www.talendbyexample.com/talend-custom-component-xsd.html>
http://www.powerupbi.com/talend/componentCreation_1.html
<https://help.talend.com/display/KB/What+is+a+Talend+component>
<https://help.talend.com/display/KB/How+to+create+a+custom+component>
<http://www.talendbyexample.com/talend-api-for-component-designers.html>
<http://bekwam.blogspot.be/2011/04/column-metadata-in-talend-open-studio.html>