talend | Connecting the Data-Driven Enterprise >

Lab Guide

ESB Basics

Version 6.2

Talend Inc.
800 Bridge Parkway, Suite 200
Redwood City, CA 94065
United States
+1 (650) 539 3200

Welcome to Talend Training

Congratulations on choosing a Talend training course.

Working through the course

You will develop your skills by working through use cases and practice exercises using live software. Completing the exercises is critical to learning!

If you are following a self-paced, on-demand training (ODT) module, and you need an answer to proceed with a particular exercise, use the help suggestions on your image desktop. If you can't access your image, contact customercare@talend.com.

Exploring

You will be working in actual Talend software, not a simulation. We hope you have fun and get lots of practice using the software! However, if you work on tasks beyond the scope of the training, you could run out of time with the environment, or you could mess up data or Jobs needed for subsequent exercises. We suggest finishing the course first, and if you have remaining time, explore as you wish. Keep in mind that our technical support team can't assist with your exploring beyond the course materials.

For more information

Talend product documentation (help.talend.com)

Talend Community (community.talend.com)

Sharing

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

CONTENTS

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

LESSON 1

Creating mediation routes

This chapter discusses the following.

Overview

This first lab introduces you to the creation of a simple mediation route. **Note:** This lab walks you through every step you need to take; however, for best results, you must have some solid experience in Talend for Data Integration.

Objectives

After completing this lesson, you will be able to:

- Create a new Talend project
- Create a new mediation route
- Use an ESB component
- Differentiate between a consumer endpoint and a producer endpoint

Next step

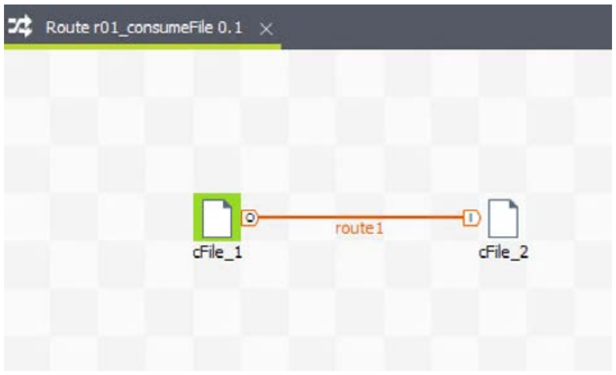
Make sure you can connect to your training Virtual Machine (VM), then move on to the first lab.

Simple file-consuming route

Overview

This lab will guide you in creating your first ESB project and mediation route.

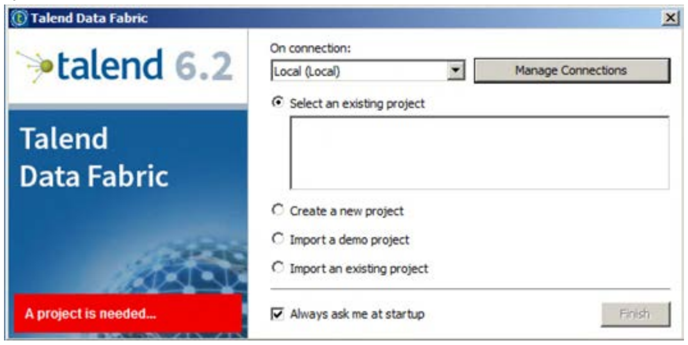
This very simple route involves two cFile components and a route. The first cFile component scans a repository for new files. When a new file is detected, it is consumed by the component and sent via the route to the second cFile component. When the second component receives a message from the route, it produces a new file similar to the one that was consumed.



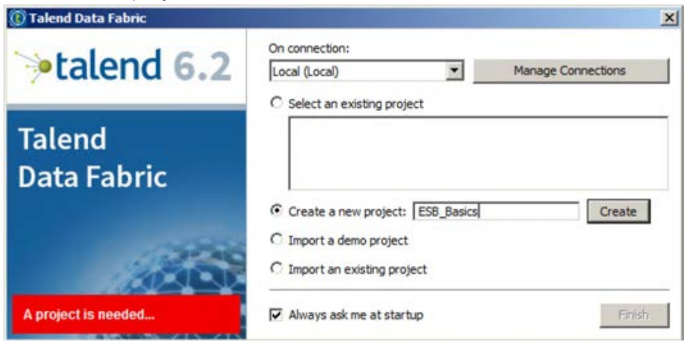
Creating a new project

The process of creating a new local ESB project is similar to that of creating a new DI project.

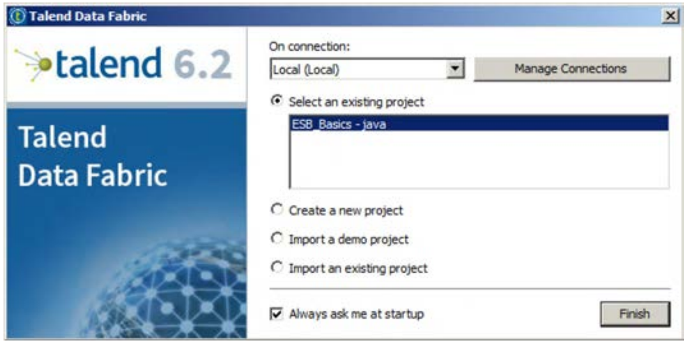
1. Double-click the **Talend Studio** icon on the desktop of your training Virtual Machine (VM). The Talend connection window opens.



2. Next to **Create a new project**, click the radio button. When the text box appears, enter *ESB_Basics* as the project name. To create the project, click the **Create** button.



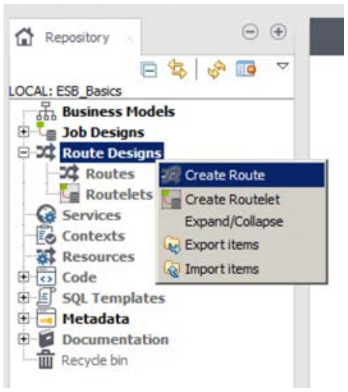
3. The project appears in the list of existing projects. To open it, click the **ESB_Basics** project, then the **Finish** button.



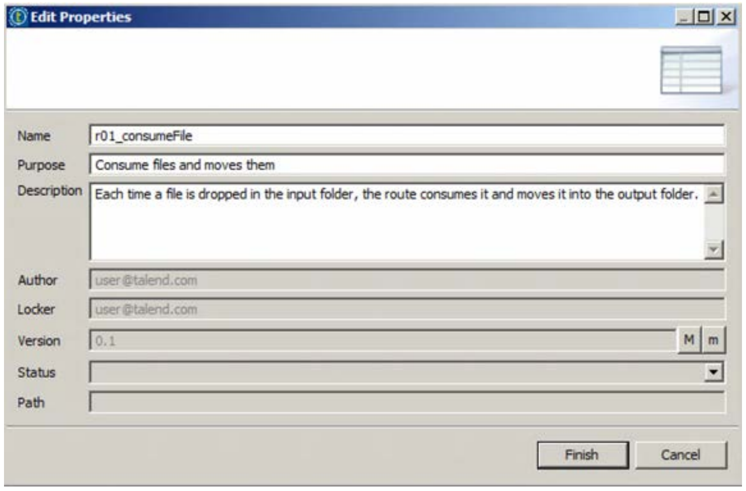
Creating a simple mediation route

When your project is open, you can create your first mediation route.

1. In the **Repository**, locate the **Route Designs** section and right-click **Route Designs**. To create a new route, in the contextual menu, click **Create Route**.

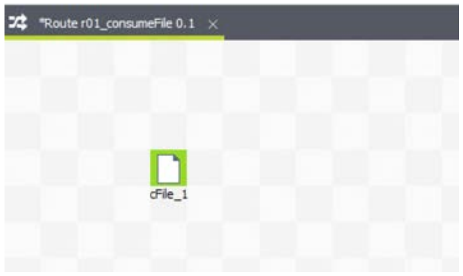


2. The **New Route** window opens. Name your route *r01_consumeFile*.
In addition to a valid title, Talend recommends that you add a purpose and description to your route. For this first route, add the following purpose:
Consume files and move them
Add a more detailed description, such as:
Each time a file is dropped in the input folder, the route consumes it and moves it to the output folder.

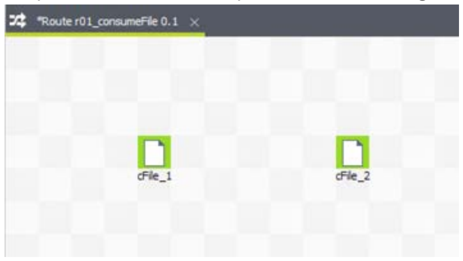


Name	r01_consumeFile
Purpose	Consume files and moves them
Description	Each time a file is dropped in the input folder, the route consumes it and moves it into the output folder.
Author	user@talend.com
Locker	user@talend.com
Version	0.1
Status	
Path	

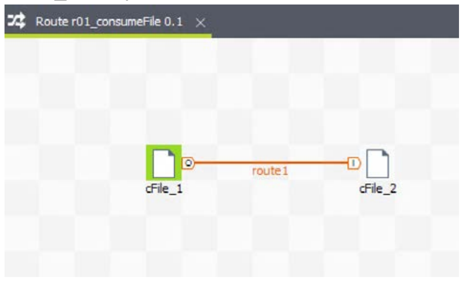
3. To create the route, click the **Finish** button.
   The new route opens in an empty design area.
4. Inspect the **Palette** on the right side of the studio. As you can see, the components are different than the ones you used when creating DI Jobs.
5. Drop a **cFile** component into the design area, either by dragging it from the **Palette** or typing *cFile* while pointing the mouse to the design area.



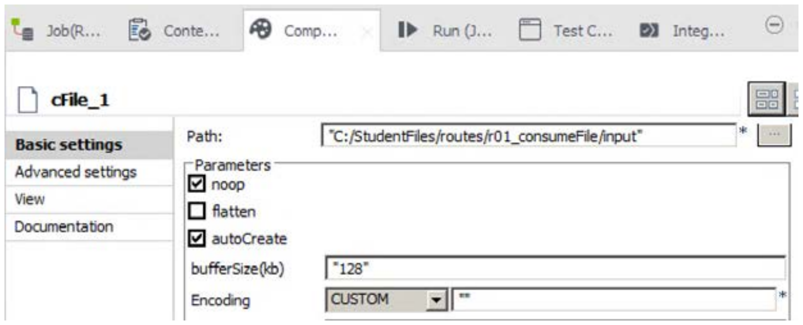
   The cFile component can read files when used as a consumer endpoint and write files when used as a producer component.
6. Drop a second **cFile** component into the design area and arrange both components as in the following screenshot:



7. To create a route between the two components, right-click the **cFile_1** component and select **Row>Route**, then click the **cFile_2** component.



   To create a simple route, you can also right-click a component and drag it to the destination component.
8. The route defines which component is the consumer and which is the producer. Here, cFile_1 is at the beginning of the route, so it is the consumer endpoint, whereas cFile_2, at the end, is the producer endpoint.
9. To configure **cFile_1**, click the component and go to the **Component** view.
10. To configure the **Path** parameter, click the **Ellipsis (...)** button and navigate to the following folder:
    C:\StudentFiles\routes\r01_consumeFile\input
    Click **OK** to validate.



Path: "C:/StudentFiles/routes/r01_consumeFile/input"

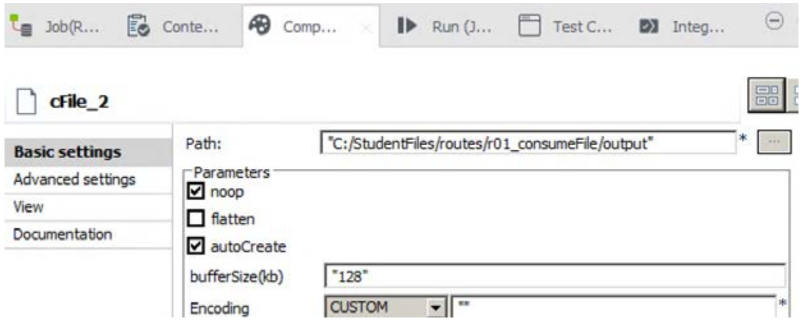
Parameters
- [✓] noop
- [ ] flatten
- [✓] autoCreate

bufferSize(kb): "128"

Encoding: CUSTOM ""

This parameter tells the component to scan the folder for new files. Each time a new file appears in this folder, the component sees it as an event and reads the file.

11. To configure **cFile_2**, click on the component and go to the **Component** view.
12. To configure the **Path** parameter, click the **Ellipsis** button and navigate to the following folder:
    C:\StudentFiles\routes\r01_consumeFile\output
    Click **OK** to validate.



Path: "C:/StudentFiles/routes/r01_consumeFile/output"

Parameters
- [✓] noop
- [ ] flatten
- [✓] autoCreate

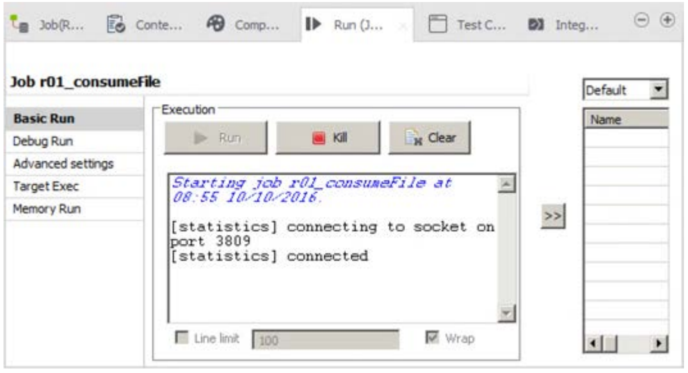
bufferSize(kb): "128"

Encoding: CUSTOM ""

This parameter tells the component to produce any message as a file in the output folder. Each time the component receives a message from the route, it creates a new file. The content of the file is the same as the message received from the route.

Running a route

1. In the studio, navigate to the **Run** view and click the **Run** button. The route is compiled and starts.

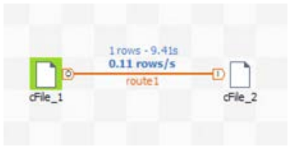


```
Starting job r01_consumeFile at
08:55 10/10/2016.

[statistics] connecting to socket on
port 3809
[statistics] connected
```

2. In a file explorer, navigate to *C:\StudentFiles\routes\r01_consumeFile*.
3. Copy the file *books.xml* and paste it in the input folder.

4. In the studio, the statistics should show that a file was consumed and processed by the route.



5. Inspect the folder *C:\StudentFiles\routes\r01_consumeFile\output*. The *books.xml* file should be there.
6. Back in the studio, do not forget to click the **Kill** button to end the route. Unlike DI Jobs, routes do not end until you terminate them.

Component options and documentation

The cFile component in the studio palette is a Talend implementation of the Camel File component. As it does for DI Jobs, Talend Studio generates Java code for you using the Apache Camel framework.

1. Click the **consumer cFile** component (**cFile_1**) and navigate to the **Component** view.
2. As you can see in the parameters, the **noop** check box is selected. To learn more about the parameters of the File component, see the official Apache Camel documentation. Component documentation is available on the Apache Camel website, http://camel.apache.org/components.html.
3. Open an Internet browser and go to the documentation page of the Apache Camel File component: http://camel.apache.org/file2.html
4. In the consumer options section, look for the **noop** parameter and learn more about this option.
5. Back in the studio, run the route and inspect the input folder.
6. Kill the route and deselect the **noop** check box next to the consumer **cFile** component (**cFile_1**).
7. Run the route one more time and inspect the input folder: turning off the **noop** option moves the input files into a camel folder once they have been consumed.

Challenge

As you can see on the Apache Camel File component documentation web page, the component has many options. Most of these options do not show on the Component view, but you can use them if needed. You can use any option in the Advanced Settings panel of the Component view.

On the documentation page, find the **delete** option and apply it to the **cFile_1** component, then run the route and notice the effect.

You have completed this lab. It is time to wrap up and move to the next chapter.

Wrap-up

In this lesson, you learned how to create a new ESB project and a mediation route.

You used your first component and created a route. You can now differentiate between a component being used as a consumer endpoint and one being used as a producer endpoint.

You learned how to find documentation about an Apache Camel component, and how to configure and use options.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

LESSON 2

Exploring messages, exchanges, and the SIMPLE language

This chapter discusses the following.

Overview

This lab introduces the concepts of Camel Exchange and Message in routes. It tells you more about how to access the different parts of a message by using the Simple Expression Language.

Objectives

After completing this lesson, you will be able to:

- Precisely define exchanges and messages, and what they are made of
- Use message headers and exchange properties
- Use expressions written in Simple
- Send a log message in a route
- Customize a log message
- Use ESB components to set a body and headers

Next step

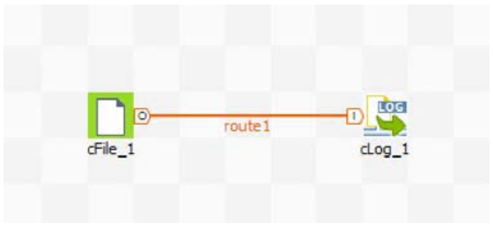
Make sure you read and understand the lesson slides, then move on to the lab instructions.

Custom Logging with Simple Expressions

Overview

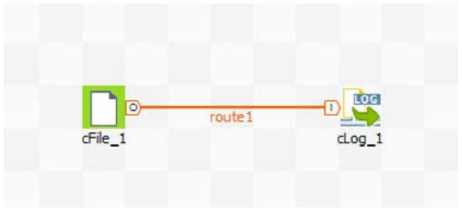
In this lab, a cFile consumer component is looking for new files in a folder. Each new file is sent via a message in an exchange along the route, then to a cLog component.

The cLog component produces a log output on the console, and displays either a standard or a custom message. This guide shows you how to integrate Simple language expressions to easily display a message header and body.

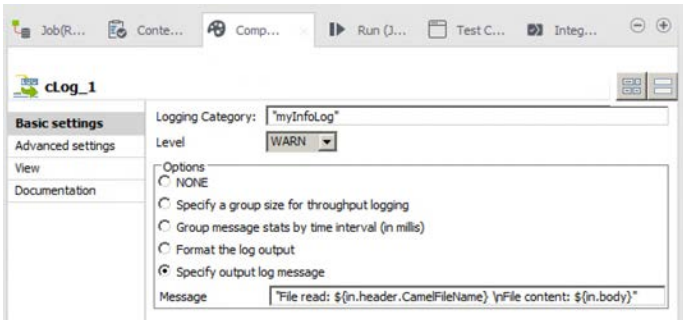


Using the cLog component

1. In Talend Studio, create a new route and name it *r02_logging*.
2. Drop a **cFile** component in the design area.
3. In the **Component** view, configure the parameter to consume from the following path:
   C:\StudentFiles\routes\r02_logging\input
4. In the design area, drop a **cLog** component to the right of the **cFile** component, then create a route from the **cFile** consumer component to the **cLog** component.

   The cLog component can send log messages on the log channel. By default, when running a route in the studio, the log channel is intercepted and displayed on the console. On a production runtime, the log channel is redirected to log files or a company logging system.
5. Select the **cLog** component, and in the **Component** view, change the **Logging Category** to *"myInfoLog"* (include the double quotes, which imply that you are providing a Java string).
   The logging category is the entity that issues the log message. Using consistent logging categories helps you quickly and easily search for and identify log messages..
6. The **Level** parameter in the **Component** view allows you to change the criticality of the log message. Keep the default WARN level. Log messages less critical than warnings are not displayed in the console.
7. Still in the **Component** view, in the **Options**, select the **Specify output log message** option. This allows you to use a custom log message in the **Message** text box.
   In the **Message** text box, enter:
   "File read: ${in.header.CamelFileName} \nFile content: ${in.body}"



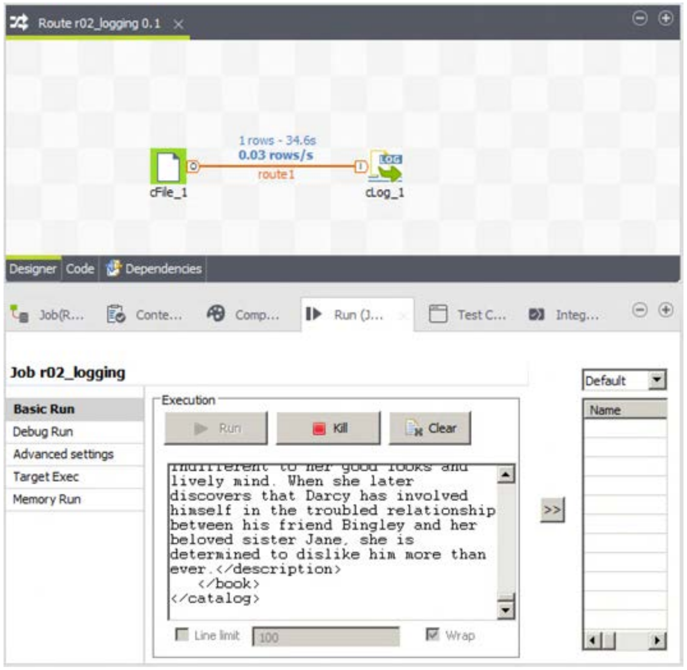
Logging Category: "myInfoLog"
Level: WARN
Options
- NONE
- Specify a group size for throughput logging
- Group message stats by time interval (in millis)
- Format the log output
- [✓ Specify output log message]

Message: "File read: ${in.header.CamelFileName} \nFile content: ${in.body}"

Here, you use two Simple expressions. The first, ${in.header.CamelFileName}, returns a header value containing the name of the file contained in the message. The second, ${in.body}, returns the body of the message, which consists of the file content read by the cFile consumer component.
Note that the header value *CamelFileName* was automatically set by the cFile component. Most components enrich their messages with header information. For each component, the list of automatically set headers can be found in the the "Message Headers" section of the Apache documentation.

Running the route

1. Go to the **Run** view of your route and click the **Run** button.
2. In a file explorer, navigate to folder *C:\StudentFiles\routes\r02_logging\*
   Find the file *books.xml*, as well as the **input** folder that your cFile consumer component is scanning.
3. Copy *books.xml* and paste it in the **input** folder.
4. The file is consumed by the cFile component and immediately sent to the cLog component. The console then shows the log message.



```
indifferent to her good looks and
lively mind. When she later
discovers that Darcy has involved
himself in the troubled relationship
between his friend Bingley and her
beloved sister Jane, she is
determined to dislike him more than
ever.</description>
   </book>
</catalog>
```

5. Do not forget to kill the route once you do not need it anymore.

Challenge

Take a few minutes to read the "Message Headers" section of the Apache documentation of the file component at http://camel.a-pache.org/file2.html.

Based on what you know, update the **`cLog`** component configuration to include a second header in the custom log message.

Proceed to the next lab to learn how to simply use components to set a message body and message headers.

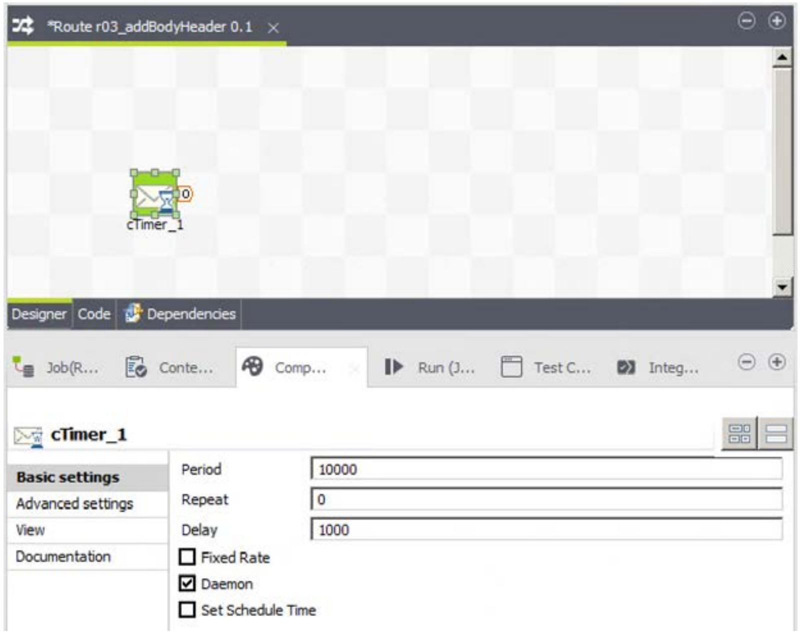
Setting a Body and Headers with ESB Components

Overview

This lab tells you about new components that let you set a new message body or message headers.

Consuming time...

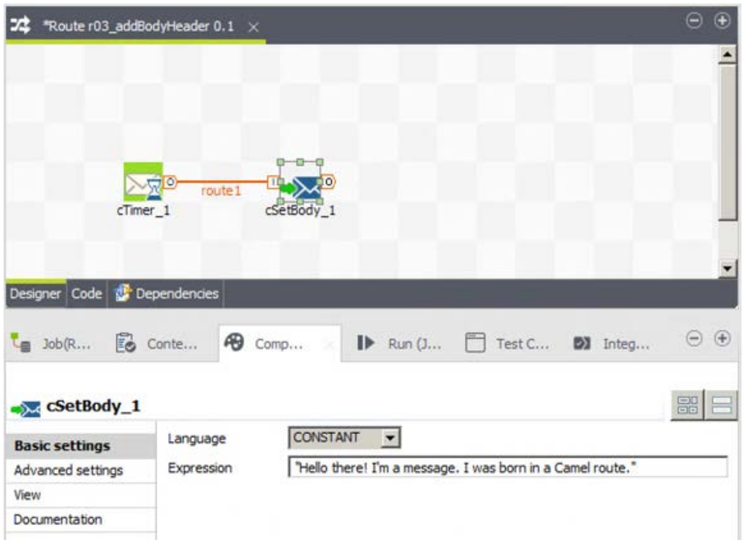
1. In Talend Studio, create a new route and name it *r03_addBodyHeader*.
2. Drop a **cTimer** component into the design area.
   The cTimer component is the consumer endpoint for this route. cTimer is configured with a time period in milliseconds. The component consumes time: each time it consumes the number of seconds set in the period parameter, it sends an exchange with an empty message along the route.
3. In the **Component** view of the **cTimer** component, set the **Period** parameter to 10000 (each time the component consumes 10 seconds (10,000 ms), it sends a message.



Period: 10000
Repeat: 0
Delay: 1000
☐ Fixed Rate
☑ Daemon
☐ Set Schedule Time

Setting body and headers

1. In the design area, drop a **cSetBody** component to the right of the **cTimer** component. This component allows you to set new body content.
   Set the body as follows:
   Language: *CONSTANT*
   Expression: *"Hello there! I'm a message. I was born in a Camel route."*
   Create a route from **cTimer** to **cSetBody**.



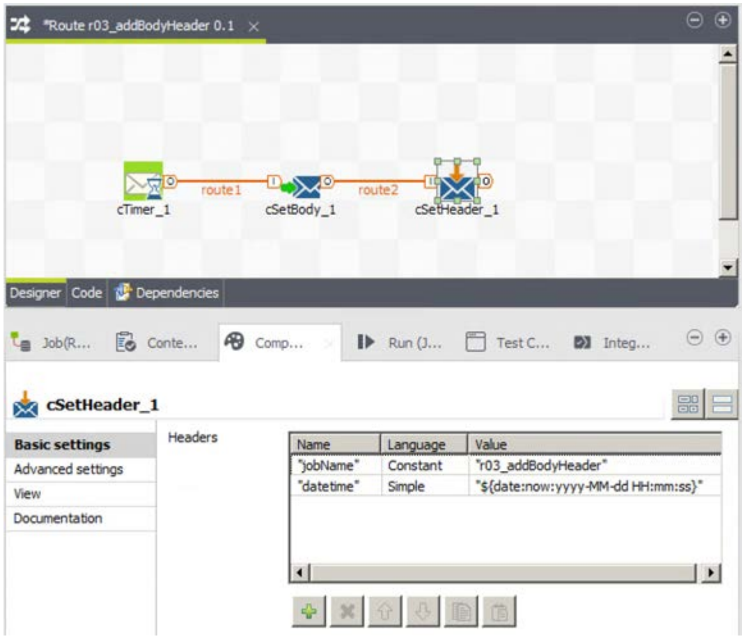
Language: CONSTANT
Expression: "Hello there! I'm a message. I was born in a Camel route."

From this component on, you set the body of the message in the **Expression** text box.

2. In the design area, drop a **cSetHeader** component to the right of the **cSetBody** component. This component allows you to create and set new header key/value pairs.
3. To create a new header, click the **+** (plus sign) button. Create two new headers and set them as follows.
   First header:
   Name: *"jobName"*
   Language: *Constant*
   Value: *"r03_addBodyHeader"*

   Second header:
   Name: *"datetime"*
   Language: *Simple*
   Value: *"${date:now:yyy-MM-dd HH:mm:ss}"*

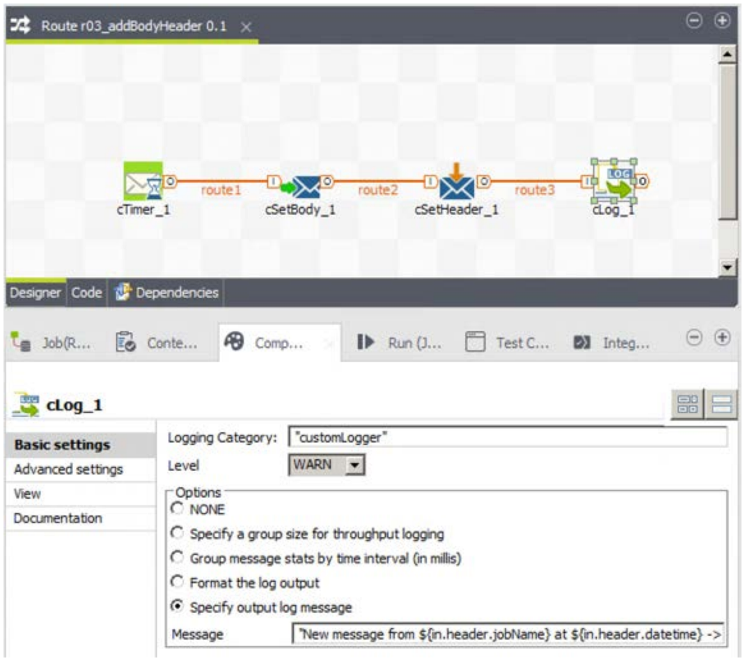
   Create a route from **cSetBody** to **cSetHeader**.



Name	Language	Value
"jobName"	Constant	"r03_addBodyHeader"
"datetime"	Simple	"${date:now:yyyy-MM-dd HH:mm:ss}"

4. Place a new **cLog** component to the right of **cSetHeader**. Rename the **Logging Category** *"customLogger"* and set the **Options** to **Specify output log message**.
   Enter the following log message:
   "New message from ${in.header.jobName} at ${in.header.datetime} ->\n ${in.body}"

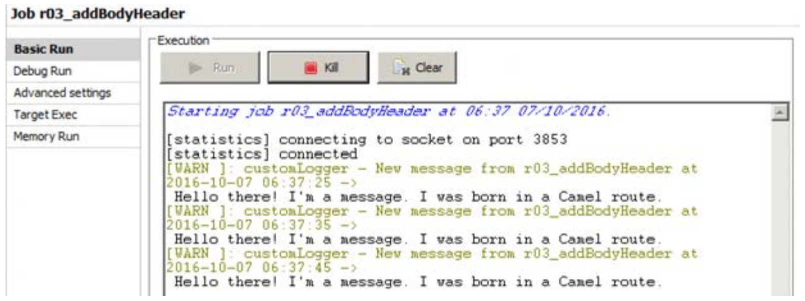
   Create a route from **cSetHeader** to **cLog**.



Logging Category: "customLogger"
Level: WARN
Options
( ) NONE
( ) Specify a group size for throughput logging
( ) Group message stats by time interval (in millis)
( ) Format the log output
(●) Specify output log message
Message: "New message from ${in.header.jobName} at ${in.header.datetime} ->

Running the route

1. Go to the **Run** view of the route and click the **Run** button.
2. Make sure that the cTimer component sends an exchange on the route every 10 seconds. You should see the result in the console.



```
Starting job r03_addBodyHeader at 06:37 07/10/2016.

[statistics] connecting to socket on port 3853
[statistics] connected
[WARN ]: customLogger - New message from r03_addBodyHeader at
2016-10-07 06:37:25 ->
 Hello there! I'm a message. I was born in a Camel route.
[WARN ]: customLogger - New message from r03_addBodyHeader at
2016-10-07 06:37:35 ->
 Hello there! I'm a message. I was born in a Camel route.
[WARN ]: customLogger - New message from r03_addBodyHeader at
2016-10-07 06:37:45 ->
 Hello there! I'm a message. I was born in a Camel route.
```

3. Do not forget to kill the route once you do not need it anymore.

You've completed this lab. It's time to wrap up and move to the next chapter.

Wrap-up

In this lesson, you learned more about the structure of exchanges and messages. Now you know that messages store their content in a body section, and store more information in the headers section. You also learned that messages are stored in an exchange that can hold one or two messages.

The first lab showed how to use a new component, called cLog, to issue log messages in the system. Thanks to the Simple language, you were able to easily access header and body information, and display it in a custom message.

The second lab taught you how to use the cSetHeader and cSetBody components to easily set a new body or new header key/value pairs.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let’s continue >**.

LESSON 3

Using Java Beans

This chapter discusses the following.

Overview

Sometimes you may need to process an exchange or message with a complex or very specific set of rules. If the Talend ESB components do not provide the complexity level you need, it is useful to know how to be able to create your own code and call it in a route to process your exchange and message.

This lab teaches you how to create simple Java classes called beans and call them in a route.

Objectives

After completing this lesson, you will be able to:

- Access an exchange and a message via Java code
- Create a Java bean class
- Pass an exchange as a method parameter
- Pass a header as a method parameter
- Call a bean in a route

Next step

A good understanding of Java is required for this lab. Good luck with this new route!

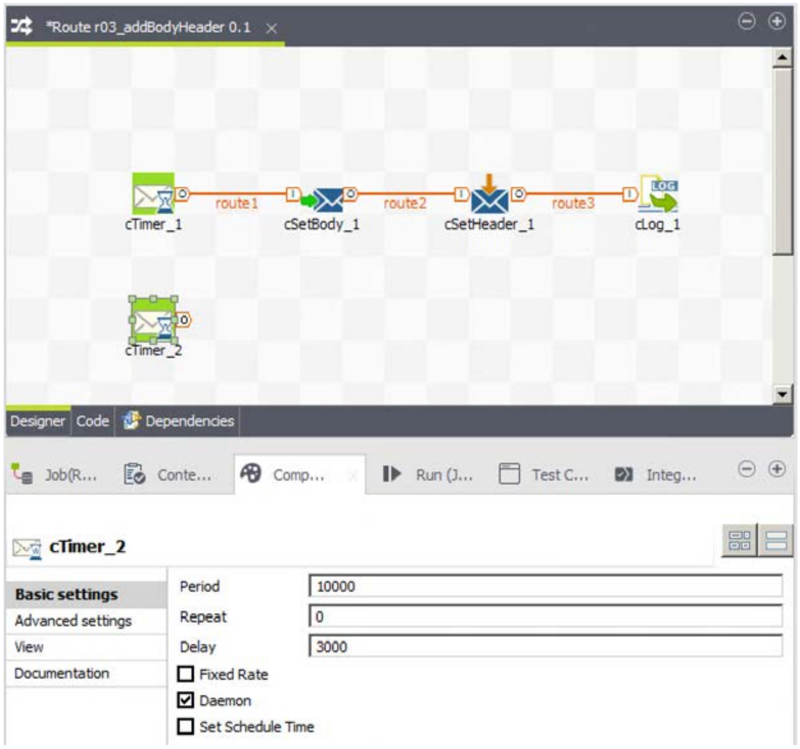
Accessing an Exchange via Java

Overview

This lab demonstrates the use of the cProcessor component, which allows you to introduce a few lines of Java code in a route.

Adding a new route

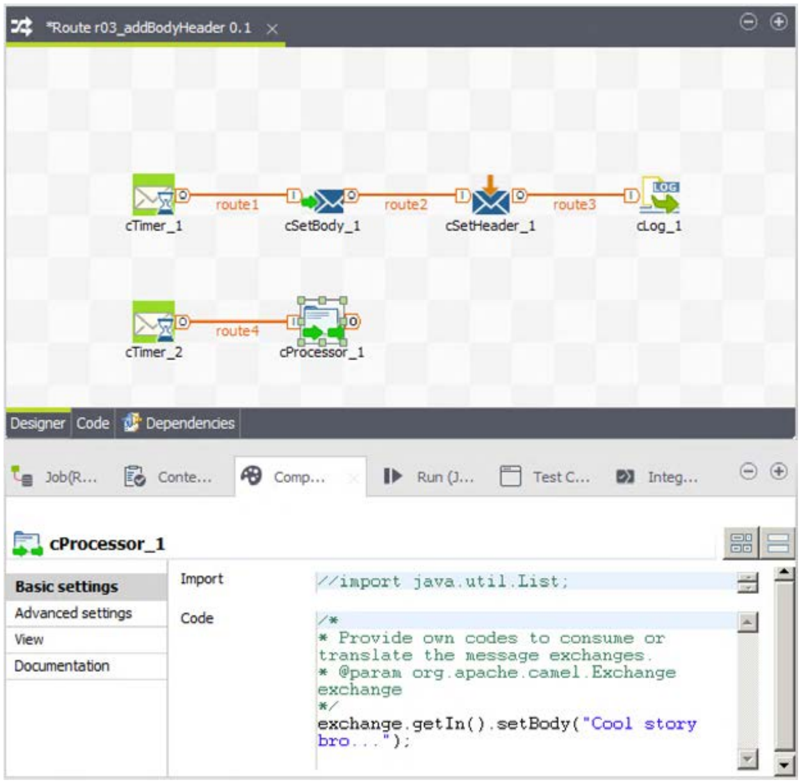
1. Open the **r03_addBodyHeader** route.
2. Under the existing route, add a new **cTimer** component. Set its **Period** to *10000* and its **Delay** to *3000*. The delay parameter is the milliseconds the component waits before starting, once the route is started.



cTimer_2

Period: 10000
Repeat: 0
Delay: 3000
☐ Fixed Rate
☑ Daemon
☐ Set Schedule Time

3. Add a **cProcessor** component to the right of the **cTimer_2** component and create a route from **cTimer_2** to **cProcessor_1**.
4. Click the **cProcessor** component and go to the **Component** view. As you can see, you can input Java code straight from the **Component** view. In the **Code** box, enter *exchange.getIn().setBody("Cool story bro...");*



cProcessor_1

Import

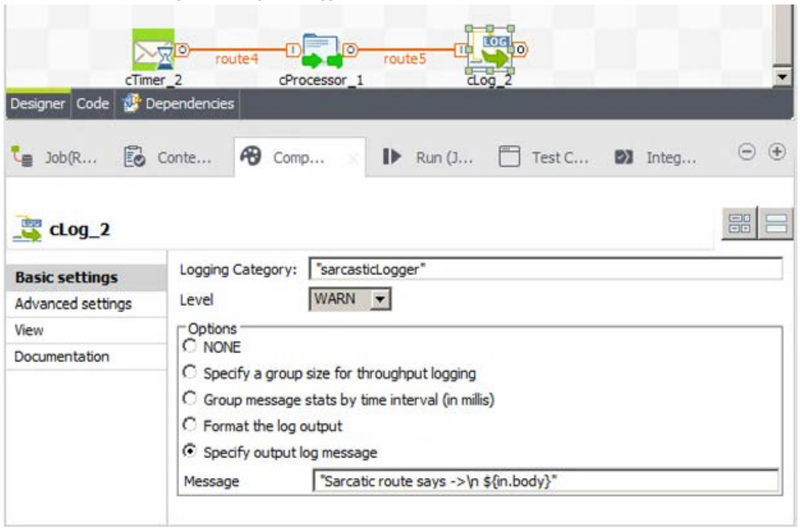
```
//import java.util.List;
```

Code

```
/*
* Provide own codes to consume or
translate the message exchanges.
* @param org.apache.camel.Exchange
exchange
*/
exchange.getIn().setBody("Cool story
bro...");
```

Exchange is the instance of the current Exchange object.
The *getIn()* method instructs action on the In message—in addition, there is a getOut() method to point to the Out message.
setBody() is the method that writes to the message body.
Do not hesitate to invoke code completion (CTRL+SPACEBAR) to explore the several methods available to the exchange object.

5. Add a **cLog** component next to the **cProcessor** component. Set the **Logging Category** to *"sarcasticLogger"* and the **Options** to *Specify output log message* with the following message:
"Sarcastic route says ->\n ${in.body}"

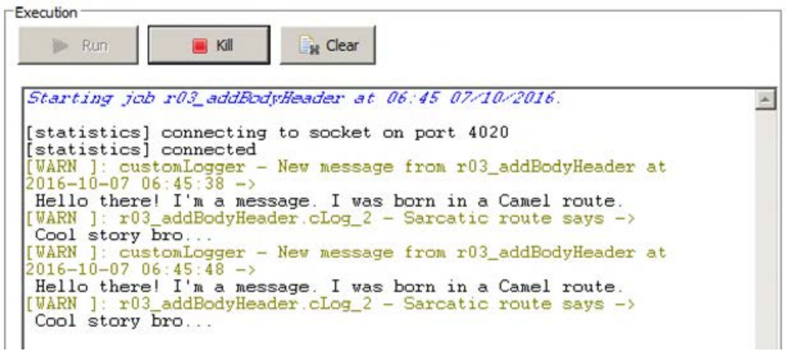


cLog_2

Logging Category: "sarcasticLogger"
Level: WARN
Options
- NONE
- Specify a group size for throughput logging
- Group message stats by time interval (in millis)
- Format the log output
- [✓ Specify output log message]

Message: "Sarcatic route says ->\n ${in.body}"

6. Run the route. You should see the two routes issuing log messages.



```
Starting job r03_addBodyHeader at 06:45 07/10/2016.

[statistics] connecting to socket on port 4020
[statistics] connected
[WARN ]: customLogger - New message from r03_addBodyHeader at
2016-10-07 06:45:38 ->
 Hello there! I'm a message. I was born in a Camel route.
[WARN ]: r03_addBodyHeader.cLog_2 - Sarcatic route says ->
 Cool story bro...
[WARN ]: customLogger - New message from r03_addBodyHeader at
2016-10-07 06:45:48 ->
 Hello there! I'm a message. I was born in a Camel route.
[WARN ]: r03_addBodyHeader.cLog_2 - Sarcatic route says ->
 Cool story bro...
```

Challenge

Using code completion, explore the methods you can access on the exchange instance and use the **cProcessor** component to write a line of Java code that adds a new header to your message. Using the Simple language, display this header in the **cLog_2** component.

The next lab will guide you in writing more-complex Java code in beans.

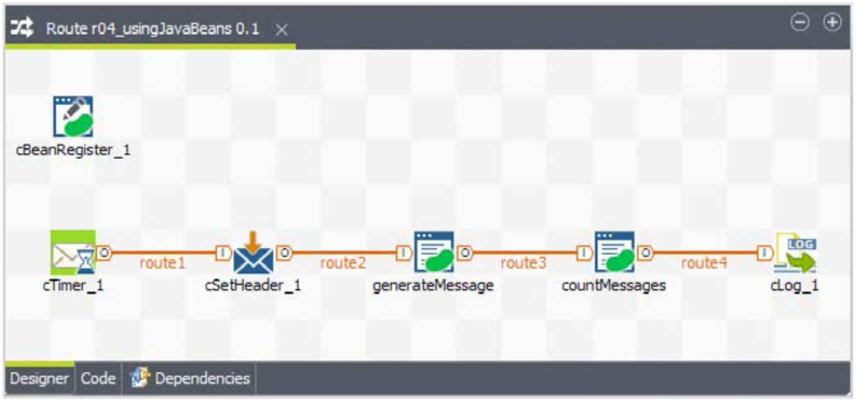
Creating a Java Bean

Overview

As you know from the previous lab, you can use Java code in a cProcessor component. However, this component reaches its limits very fast. It is not adapted for long and complex blocks of code, and you cannot reuse it.

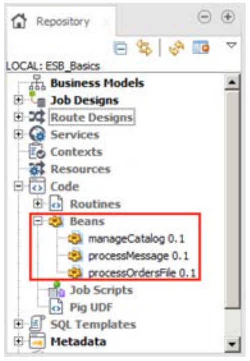
In this lab, you will create a more complex Java code and store it in a simple Java class—a Java bean—that you can call from a route.

The route you are about to create starts with a cTimer component. Every second, it sends a blank exchange along the route. A cSetHeader component sets a header variable named *company*. Then, a cBean component calls a bean class and method. This method takes two parameters: the exchange itself and the header that was just set. Based on these parameters, using the value of the header, the method writes a new message body in the exchange. A second bean method is then called to count the number of processed messages. The route ends with a cLog component and displays the message in the console.



Creating a new bean

Beans are stored in the repository, in Code > Beans.



1. To create a new bean, in the **Repository**, expand **Code** and right-click **Beans**. On the contextual menu, choose **Create Bean**.
2. In the **New Bean** window, name the bean *processMessage* and click **Finish**.
3. A code editor window opens with the skeleton of your bean class.
4. In a file explorer, go to *C:\StudentFiles\routes\r04_usingJavaBeans* and use Notepad++ to open *processMessage.txt*.
    a. In the **Imports** section, note that you need to import the Apache Camel classes of the exchange and message in order to access the appropriate methods.
    b. The first method, *updateMessageBody*, takes two parameters. The exchange is passed by creating it in the parameters; in this instance, it is named *exch*. The header *company* is passed by using the Java annotation defined in the class of the message object, *@Header*. To get a parameter from a header, you must use the following syntax: *@Header("name_of_header") type paramName*

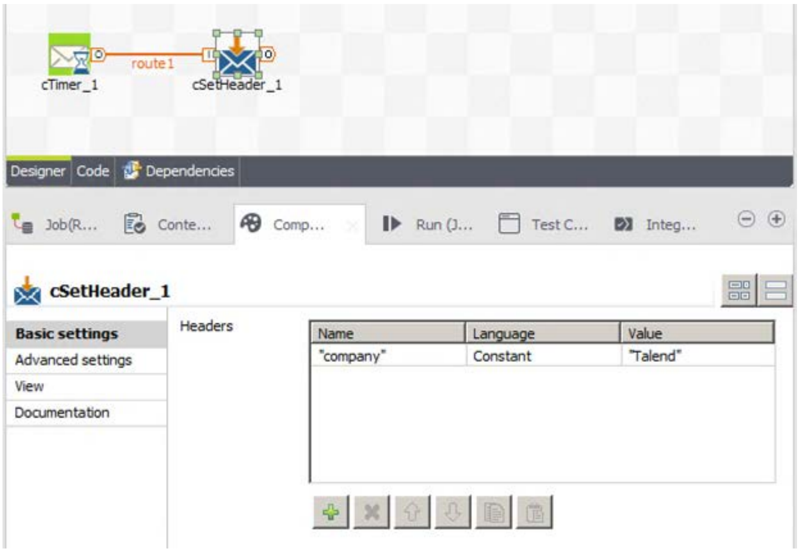
c. The *updateMessageBody* method does not return anything; it acts on the exchange itself. In this instance, using the Java instructions you used in the latest lab, it randomly chooses a message and writes it in the body.

d. The second method is called *incrementMessages*. Each time it is called, it increments the static counter variable *nbOfMsg*.

5. Copy the content of the file *processMessage.txt* and paste it in the *processMessage* code in the studio. Make sure this new code replaces the previous pregenerated code skeleton.
6. Save the bean and close its window.

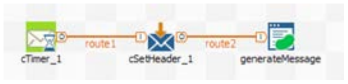
Invoking a bean

1. Create a new route and name it *r04_usingJavaBeans*.
2. Add a new **cTimer** component. Leave all parameters in their default values.
3. Next to the **cTimer** component, add a new **cSetHeader** component. Create a route from **cTimer_1** to **cSetHeader_1**.
4. In the **cSetHeader** component, add a new header and configure it as follows:
Name: *"company"*
Language: *Constant*
Value: *"Talend"*



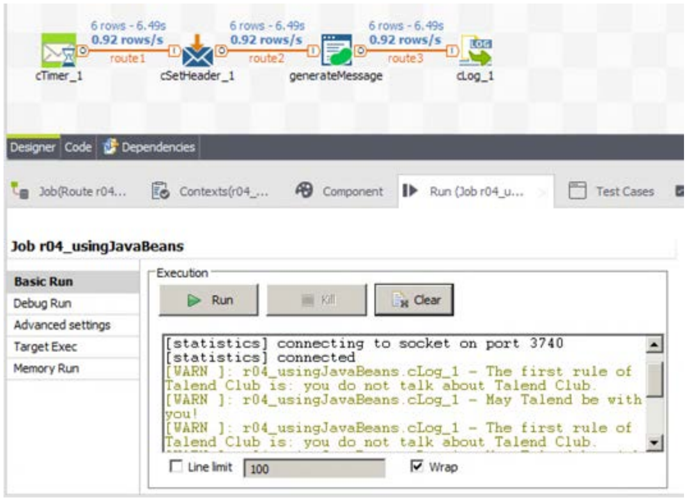
Name	Language	Value
"company"	Constant	"Talend"

5. Drop a **cBean** component to the right of the **cSetHeader** component and name it *generateMessage*.



6. Click the **cBean** component, and, in the **Component** view, select the **New Instance** radio button. This option tells the component to create a new instance of the class each time the cBean component is called.
In the **Bean class** text box, enter the name of the class you want to call: *beans.processMessage.class*.
Select the **Specify the method** check box. This option allows you to choose which method to call in the bean. In the text box, enter *"updateMessageBody"*
7. Next to the **cBean** component, add a new **cLog** component. Specify the output log message so that it displays the body of the message.

8. Run the route. Every second, it should display a message generated by the bean method.



Job r04_usingJavaBeans

```
[statistics] connecting to socket on port 3740
[statistics] connected
[WARN ]: r04_usingJavaBeans.cLog_1 - The first rule of
Talend Club is: you do not talk about Talend Club.
[WARN ]: r04_usingJavaBeans.cLog_1 - May Talend be with
you!
[WARN ]: r04_usingJavaBeans.cLog_1 - The first rule of
Talend Club is: you do not talk about Talend Club.
```

☐ Line limit 100 ☑ Wrap

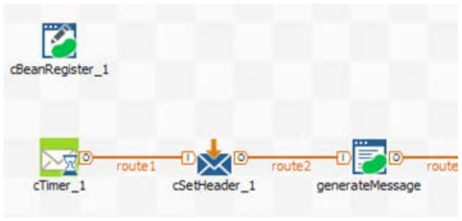
Calling a bean reference

The cBean component used in the previous example was set to "*New instance*", which means a new instance of the bean class is created each time the cBean component is invoked, and it is destroyed right after processing the exchange.

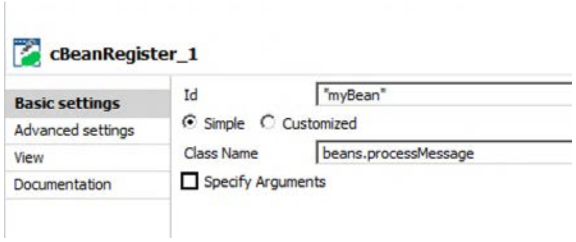
In some cases, you may need to retain information in the memory of your bean instance, which is why you can also create an object that you can call in a cBean as a reference. A reference is a name that points to a unique instance of your bean, stored in the bean registry of the route.

In the next example, you will call the second method of the bean class, *incrementMessages*. This method accesses an internal variable and increments it each time it is called. Obviously, you need to call the same instance of the object, as calling a new instance each time you invoke the bean resets the counter for each new call.

1. In the route **r04_usingJavaBeans**, add a **cBeanRegister** component.

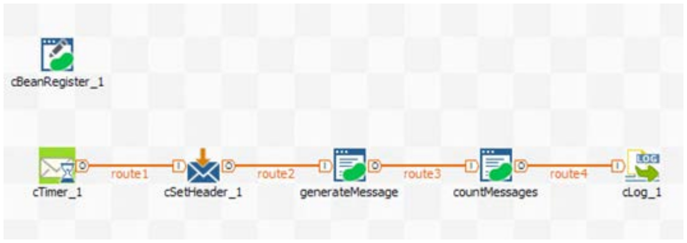


2. In the **Component** view of this new component, enter *"myBean"* as the **Id** and *beans.processMessage* as the **Class Name**. This creates an instance of your class *processMessage*; its reference is **myBean**.

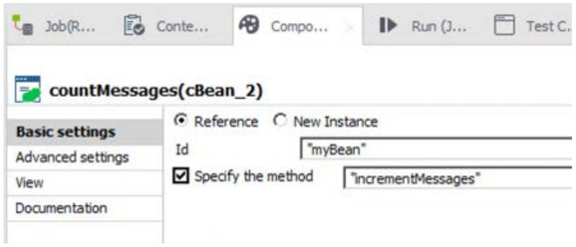


Id "myBean"
[✓ Simple] Customized
Class Name beans.processMessage
☐ Specify Arguments

3. Between the **generateMessage** and **cLog_1** components, add a new **cBean** component and name it *countMessages*.



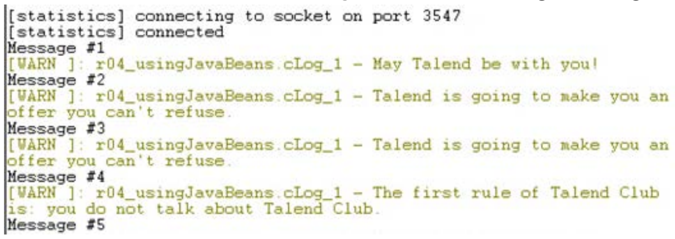
4. To call a bean reference, in the **Component** view of *countMessages*, select the **Reference** radio button. Set the **Id** to *"myBean"*, select the **Specify the method** check box, and set the method to *"incrementMessages"*



countMessages(cBean_2)

[✓ Reference] New Instance
Id "myBean"
[✓] Specify the method "incrementMessages"

5. Run the route. The *incrementMessages* method should log a message for each new message passing through it.



```
[statistics] connecting to socket on port 3547
[statistics] connected
Message #1
[WARN ]: r04_usingJavaBeans.cLog_1 - May Talend be with you!
Message #2
[WARN ]: r04_usingJavaBeans.cLog_1 - Talend is going to make you an
offer you can't refuse.
Message #3
[WARN ]: r04_usingJavaBeans.cLog_1 - Talend is going to make you an
offer you can't refuse.
Message #4
[WARN ]: r04_usingJavaBeans.cLog_1 - The first rule of Talend Club
is: you do not talk about Talend Club.
Message #5
```

You have completed this lab. It is time to wrap up and move to the next chapter.

Wrap-up

In this lab, you learned how to access the exchange and its elements by using Java code. You know how to introduce a short block of code in a route by using a cProcessor component, and you can create more-complex methods in a Java bean class.

You can use the cBean component to invoke one of your beans, as well as differentiate between calling a new instance and calling a reference.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let’s continue >**.

LESSON 4

Rerouting Messages

This chapter discusses the following.

Overview

This lab shows you how to use the cDirect component as the consumer and producer endpoint to send a message from one route to another.

Objectives

After completing this lesson, you will be able to:

- Define a Camel context
- Route a message from one route to another

Next step

Read the lesson slides, then start the lab.

Using the cDirect Component to Reroute a Message

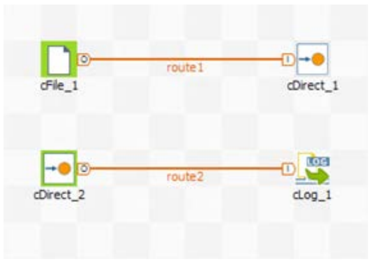
Overview

This lab illustrates the concept of message rerouting with a very simple example: two routes run in the same Camel context.

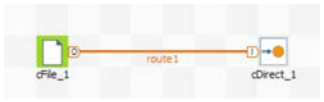
The first route consumes files from an input file and sends them to a cDirect producer component. The cDirect component is the Talend implementation of the Apache Camel direct component in the palette.

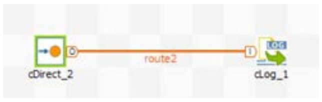
The second route starts with a cDirect consumer component and sends its message to a cLog component.

The cDirect producer from the first route and the cDirect consumer from the second route use the same URI. This means that when the first route produces a message on the URI, the same message is consumed by the cDirect component of the second route.



Using and configuring cDirect components

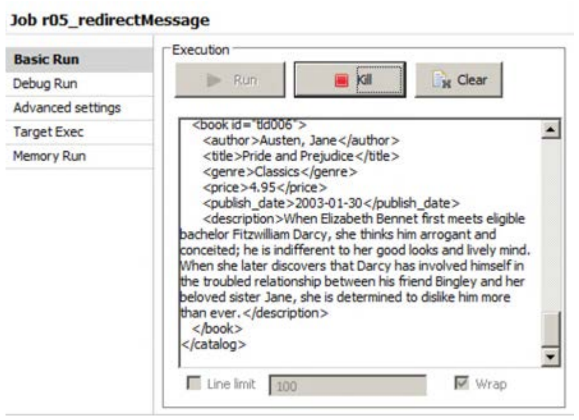
1. In Talend Studio, create a new route and name it *r05_redirectMessage*.
   Now that you know more about Camel contexts, you understand that when you create a new route in the Studio repository, you actually create a Camel context. Since "context" already represents a feature in the Talend repository, we use "route" to loosely describe a Camel context.
2. In *r05_redirectMessage*, create a first route
   a. Place a **cFile** component in the design area and configure it to scan the directory *C:\StudentFiles\routes\r05_redirectMessage\input*
   b. Add a **cDirect** component to the right of the **cFile** component. You will configure it later.
   c. Create a route from the **cFile** component to the **cDirect** component
   

3. Create a second route:
   a. Under the first route, add a **cDirect** component.
   b. Add a **cLog** component to the right of the **cDirect** component. Configure it so the log message displays the body of the message *"${in.body}"*.
   c. Create a route from the **cDirect** component to the **cLog** component.
   

4. Click the **cDirect_2** component. In the **Component** view, in the **Name** text box, enter *"mySecondRoute"*.
   The direct component consumes any message arriving on the URI *direct:mySecondRoute*.
5. Click the **cDirect_1** component. Because this component is used as a producer endpoint, the configuration available in the **Component** view is different: it displays a list of all available cDirect consumer endpoints. Select the **cDirect_2** option. This interface simplifies the way you use the cDirect components: you tell the component that the message should be rerouted to the same URI that cDirect_2 is listening to as a consumer.

Running the route

1. Go to the **Run** view of the route and click the **Run** button.
2. In a file explorer, go to *C:\StudentFiles\routes\r05_redirectMessage*. Copy the file *books.xml* to the **input** folder.
3. Make sure the message travels the first route and is rerouted to the second route. You should see the file content logged in the console.



```
  <book id="bk106">
    <author>Austen, Jane</author>
    <title>Pride and Prejudice</title>
    <genre>Classics</genre>
    <price>4.95</price>
    <publish_date>2003-01-30</publish_date>
    <description>When Elizabeth Bennet first meets eligible
bachelor Fitzwilliam Darcy, she thinks him arrogant and
conceited; he is indifferent to her good looks and lively mind.
When she later discovers that Darcy has involved himself in
the troubled relationship between his friend Bingley and her
beloved sister Jane, she is determined to dislike him more
than ever.</description>
  </book>
</catalog>
```

4. Do not forget to kill the route when you are finished.

You have completed this lab. It is time to wrap up and move to the next chapter.

Wrap-up

In this chapter, you learned more about Camel contexts, and you know that they are containers for all the elements that build routes.

You also used the cDirect component to reroute a message from one route to another in the same Camel context.

In a future chapter, you will see how a message broker can help you exchange messages between two different Camel contexts.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let’s continue >**.

LESSON 5

Discovering EIPs

This chapter discusses the following.

Overview

Enterprise Integration Patterns (EIP) are a very important feature of the Camel routing engine. This lesson introduces you to four of the most used EIPs in application integration.

Objectives

After completing this lesson, you will be able to:

- Route a message based on its content
- Split a single message into several by using an XPath query
- Route a message through a sequence of processing routes
- Broadcast a message to several routes at the same time

Next step

When you are ready, move on to the first lab in this lesson.

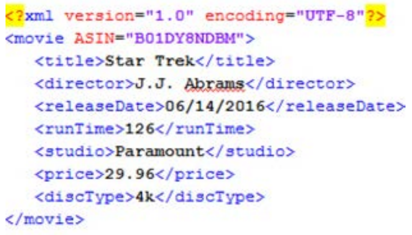
Content-based Routing

Overview

Content-based routing routes a message according to its content rather than an explicitly specified destination.

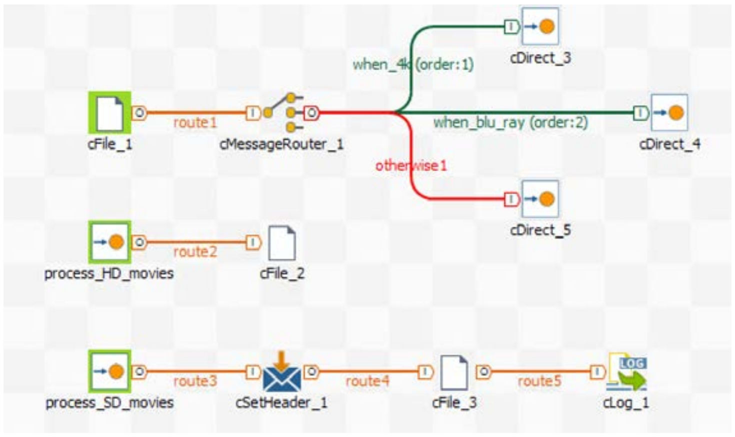
The cMessageRouter EIP allows you to apply conditions or a set of rules to define a destination.

In this lab, you need to process XML files. The files,part of a global movie catalog, each hold information about one movie disc.



```
<?xml version="1.0" encoding="UTF-8"?>
<movie ASIN="B01DY8NDBM">
   <title>Star Trek</title>
   <director>J.J. Abrams</director>
   <releaseDate>06/14/2016</releaseDate>
   <runTime>126</runTime>
   <studio>Paramount</studio>
   <price>29.96</price>
   <discType>4k</discType>
</movie>
```

You must route these files according to their disc type. When the disc type is "Blu ray" or "4k", the file must be routed to an "HD" folder. Otherwise it must be routed to an "SD" folder, in which case, a log message should warn you that *"The following movie: (movieTitle) is only available in standard definition, please consider upgrading to HD."*

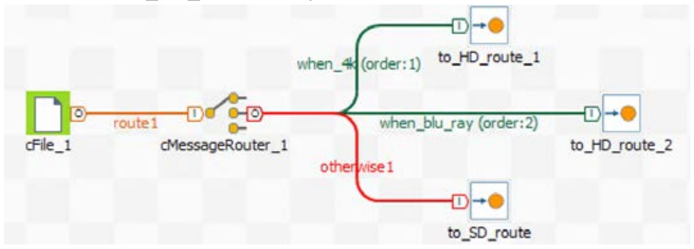


Content-based routing is determined at the cMessageRouter component level.

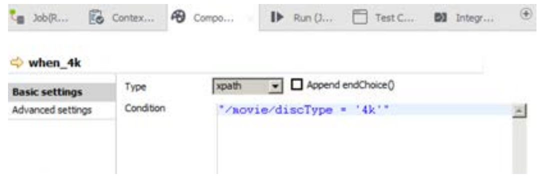
Creating routes

1. Create a new Camel context (a "route", as it is known in relation to the repository) and name it *r06_messageRouter*.
2. In *r06_messageRouter*, create the first route:
   a. Place a **cFile** component in the design area and configure it to scan the following directory: *C:\StudentFiles\routes\r06_redirectMessage\input*
   b. Add a **cMessageRouter** component to the right fo the **cFile** component. As you can see in its **Component** view, there is no parameter for this component.
   c. Place three **cDirect** components after the **cMessageRouter** component. These are destinations for the three conditions defined after the **cMessageRouter** component.
   d. Rename these **cDirect** components *to_HD_route_1*, *to_HD_route_2*, and *to_SD_route*.
   e. Right-click the **cMessageRouter** component. On the contextual menu, choose **Trigger > When**. To create a new link, click the **cDirect 'to_HD_route_1'** component. Rename the link *when_4K*.
   f. Right-click the **cMessageRouter** component. On the contextual menu, choose **Trigger > When** and link it to the **cDirect 'to_HD_route_2'** component. Rename the link *when_blu_ray*.

g. Right-click the **cMessageRouter** component. On the contextual menu, choose **Trigger > Otherwise** and link it to the **cDirect 'to_SD_route'** component.



h. Click the **when_4k** link and go to the **Component** view.
You can set a content-based condition using one of the languages in the drop-down menu. When the condition is true, the message is routed to the destination.
You are processing an XML file, so you use the XPath language to write your condition. On the **Type** menu, select **XPath**. In the **Condition** text box, enter *"/movie/discType = '4k'"*.



```
"/movie/discType = '4k'"
```

i. Click the **when_blu_ray** link and go to the **Component** view.
On the **Type** menu, select **XPath**. In the **Condition** text box, enter *"/movie/discType = 'Blu-ray'"*.

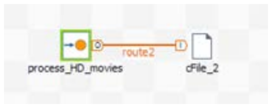
j. The '**otherwise**' link does not need configuration; any message that does not match any of the cMessageRouter conditions is routed to this destination.

3. Create the second route to process the "HD movie" messages:

a. Add a **cDirect** component and name it *process_HD_movies*. In the **Component** view, in the **Name** text box, enter *"HD"* .

b. Add a **cFile** component to the right of the **cDirect process_HD_movies** component. Set its **Path** parameter to *"C:/StudentFiles/routes/r06_messageRouter/output/HD"*.

c. Create a route from the **cDirect process_HD_movies** component to the **cFile** component.



d. In the first route, click the **cDirect** component named **to_HD_route_1**. In its Component view, in the **Use existing cDirect** menu, choose the **process_HD_movies** element.

e. Finally, click the **cDirect** component named **to_HD_route_2**and redirect it also to the existing **process_HD_movies** element.

3. Create a third route to process the "SD movie" messages:

a. Add a **cDirect** component and name it *process_SD_movies*. In the **Component** view, in the **Name** text box, enter *"SD"*.

b. To be able to display the movie title in the log message, use a **cSetHeader** component and an **XPath** query to retrieve the title from the file content. Add a **cSetHeader** component to the right of the **cDirect 'process_SD_movies'** component.

c. Configure the **cSetHeader** component as follows:

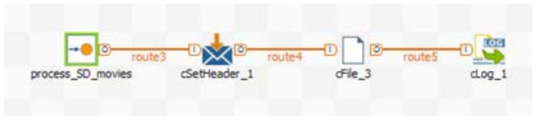
Name: *movieTitle*

Language: *XPath*

Value: *"/movie/title/text()"*

d. Add a **cFile** component next to the **cSetHeader** component. Set its **Path** parameter to *"C:/StudentFiles/routes/r06_messageRouter/output/SD"*.

e. Add a **cLog** component right after the **cFile** component and set its custom message to *"The following movie: ${in.header.movieTitle} is only available in standard definition, please consider upgrading to HD."*



f. Finally, click the **cDirect** component named **to_SD_route**and redirect it also to the existing **process_SD_movies** element.

4. Run the Camel context.
5. In a file browser, go to *C:\StudentFiles\routes\r06_messageRouter* and copy the *movieX.xml* files to the **input** folder.
6. To make sure that the messages were routed to the appropriate destination, inspect the **output folder** and log messages.

You have completed this lab. If your session is running as expected, you can move on to the next EIP.

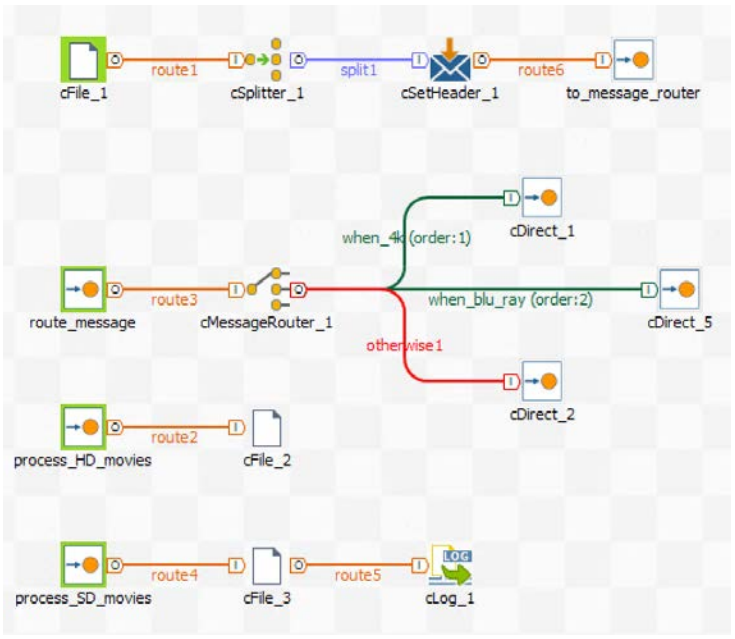
Splitting a Message

Overview

Many messages passing through an integration solution consist of multiple elements. For example, an order placed by a customer consists of more than just a single line item. As outlined in the previous lab with the cMessageRouter content-based routing, each line item may need to be handled by a different inventory system. In other words, you need to process a complete order but treat each item individually.

The cSplitter EIP breaks out a message containing a list of repeating elements, each of which can be processed individually.

In this lab, you will process an XML catalog file that contains several movie items. The goal is to process each item individually using the same process that you developed in the previous lab.

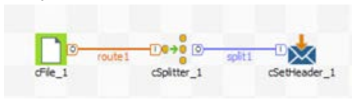


Configuring the cSplitter component

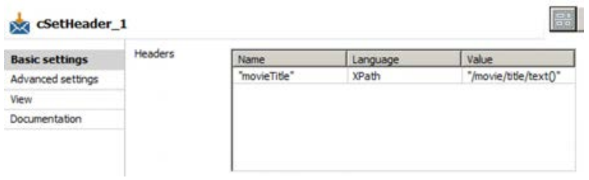
1. In the **Repository**, right-click **r06_messageRouter**, and, on the contextual menu, choose **Duplicate**. Name the new route *r07_messageSplitter*.
2. In *r07_messageSplitter*, add a new route:
    a. Place a **cFile** component in the design area and configure it to scan the following directory: *C:\StudentFiles\routes\r07_messageSplitter\input*
    b. Next to the new **cFile** component, add a **cSplitter** component. In its **Component** view, provide a rule to split the incoming message.
    c. With Notepad++, open the file *movies.xml* in *C:\StudentFiles\routes\r07_messageSplitter*. As you can see, its structure has a root tag, *<catalog>,* under which the *<movie>* tag is looping for each movie item.
    The file needs to be split at the */catalog/movie* level.
    d. Back in the studio, in the **cSplitter Component** view, select **XPath** as the **Language** for your split expression. In the **Expression** text box, enter *"/catalog/movie"*. Now the component knows where to cut your message.
    e. Create a route from the **cFile** component to the **cSplitter** component.
    f. Remember what the remainder of the process is supposed to do: messages, each containing a single movie after the cSplitter component, are to be processed by the cMessageRouter component. Based on message content, a message is routed to either the *"HD"* or *"SD"* destination. Then the message is produced as a file in a specific folder. In the previous lab, each file had its own file name, so when the final cFile produced the message as a file in an

output folder, it used the original file name. In the new process, the filename for all split messages is the same, as it comes from the same *movies.xml* file.
To fix this problem, you will extract the movie title in each message and store it in a header variable. When the message arrives at its destination, the cFile component can be configured to name the produced file after the title in the header variable.
To the right of the **cSplitter** component, add a new **cSetHeader** component.

g. Right-click the **cSplitter** component and select **Row > split** from the menu. Link the new route to the **cSetHeader** component.

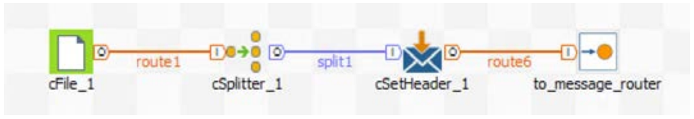


h. Configure a new header variable named *"movieTitle"* that takes the movie title as a value. If you need a hint, see the screenshot for this step.

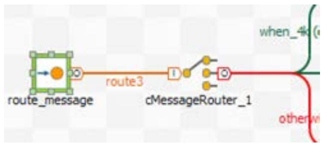


Name	Language	Value
"movieTitle"	XPath	"/movie/title/text()"

i. To send the message from this route to the **cMessageRouter** route, add a **cDirect** component after the **cSetHeader** component, and link them with a route. Rename this **cDirect** component *to_message_router*.

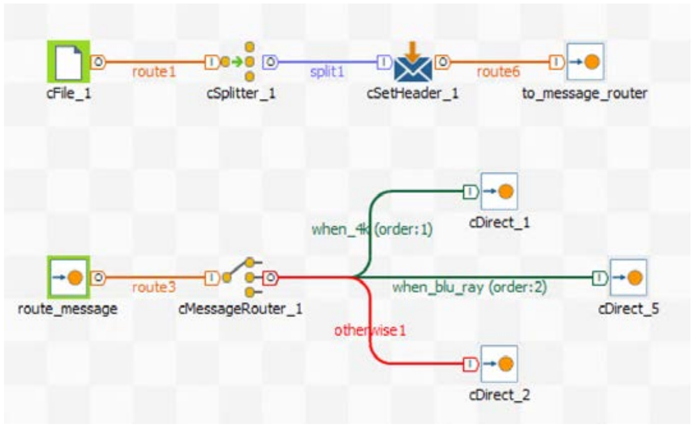


3. Now you just need to connect the routes:
   a. Delete the original **cFile** component that was placed before the **cMessageRouter** component.
   b. Place a **cDirect** component where the original **cFile** was and name it *route_message*.
   c. In the **cDirect 'route_message' Component** view, in the **Name** text box, enter *"messageRouter"*
   d. Create a route from the **cDirect 'route_message'** component to the **cMessageRouter** component.

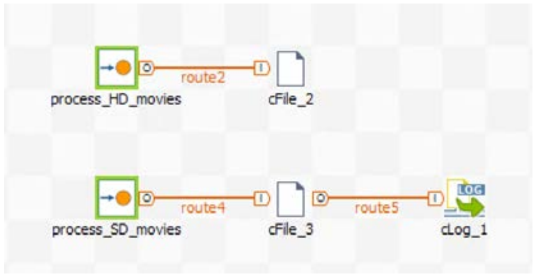


   e. Finally, in the **cSplitter** route, click the **cDirect 'to_message_router'** producer component and go to its **Component** view. In the **Use existing cDirect** parameter menu, select **cDirect - route_message**.

Your **cSplitter** route is configured to reroute messages to the **cMessageRouter** route.



4. Last, you need to adapt the two final routes so that the cFile producer components write their messages in the appropriate destinations with the appropriate file names.
    a. In the **process_HD_movies** route, select the **cFile** component and navigate to its **Component** view.
    b. In the **Path** text box, enter *"C:/StudentFiles/routes/r07_messageSplitter/output/HD"*.
    c. In the **fileName** text box, enter *"${in.header.movieTitle}.xml"*. This Simple expression names the file after the header variable *movieTitle*, and adds the .xml extension to it.
    d. In the **process_SD_movies** route, select the cFile component and navigate to its **Component** view.
    e. In the **Path** text box, enter *"C:/StudentFiles/routes/r07_messageSplitter/output/SD"*
    f. Configure the **fileName** the same way you did in Step c.
    g. Feel free to remove the **cSetHeader** component from this route; it is no longer useful because the same header is defined in the cSplitter route.



5. Run the Camel context.
6. In a file browser, go to *C:\StudentFiles\routes\r07_messageSplitter* and copy the *movies.xml* files to the **input** folder.
7. Inspect the **output** folder and log messages to make sure the messages were routed to the appropriate destination and that the files were renamed with their movie titles.

You have completed this lab. If your session is running as expected, move on to the next EIP.

Routing a Message to a Sequence of Destinations

Overview

Sometimes you may need to route a message not just to a single route, but through a whole series of processing routes.

The cRoutingSlip EIP allows you to route a message consecutively through a series of processing steps when the sequence is not known at design time and may vary for each message.

You need to attach a routing slip header variable to each message, specifying the sequence of processing steps.

The cRoutingSlip component computes the list of required steps for each message. It then starts the process by routing the message to the first processing step. After successful processing, each processing step views the routing slip and passes the message to the next processing step specified in the routing table.

In this lab, you must process an XML order file. Each item in the order has a price tag and a discount tag. If the item has no discount, it is processed as follows:

Send the warehouse a request to send the product

Bill the customer

If the item has a discount, the process has one more step:

Send the warehouse a request to send the product

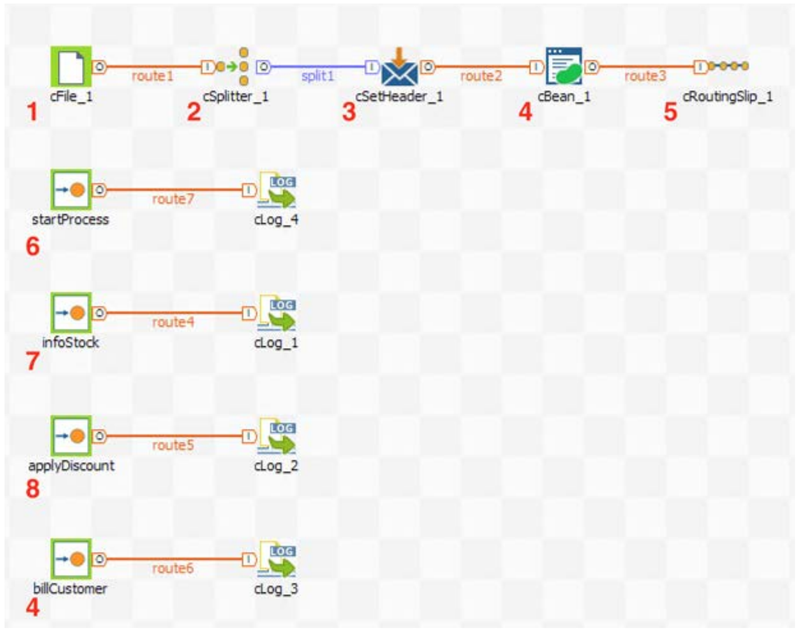
Apply the discount

Bill the customer

Each step consists of one route. Each order item is processed by a cBean to determine its sequence of processing steps. The sequence is stored in a header variable that lists all the message destinations. When the message reaches the cRoutingSlip component, the component reads the header variable and routes the message to the list of destinations in the order of the sequence.

Understanding the cRoutingSlip EIP

Before building your Camel context, make sure you understand the logic behind the cRoutingSlip EIP:



1. The cFile component reads the order file. This is a global XML file with multiple items.
2. The original message is split per item.
3. For each item, you create two header variables:

   discount: contains the value of the discount tag in the message

   oID: contains the orderID of the message

4. The message is processed by a cBean:

   If the discount value is 0, the bean method creates a header variable named *routeSequence* with the value "direct:startProcess, direct:infoStock, direct:billCustomer"

   If the discount value is greater than 0, it creates a header variable named routeSequence with the value "direct:startProcess, direct:infoStock, direct:applyDiscount, direct:billCustomer"
5. The cRoutingSlip component receives the message. It reads the routeSequence header variable and considers the comma a separator character to establish the list of destinations. Then it sends the message to the first destination (direct:startProcess). When the message reaches the end of this first route, the cRoutingSlip component routes it to the next destination, and so on until the end of the sequence.
6. The startProcess route sends a log to indicate starting to process one item.
7. The infoStock route fakes a request to the warehouse and a stock update.
8. The applyDiscount route fakes applying the discount when it is called.
9. The billCustomer route fakes editing the final bill before charging the customer.

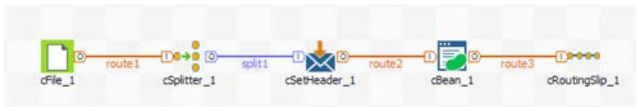
Implementing the cRoutingSlip component

1. Create a new Camel context and name it *r08_routingSlip*.
2. Start by creating the first route:
   a. Add a **cFile** component and configure it to consume in the folder *"C:/StudentFiles/routes/r08_routingSlip/input"*
   b. Add a **cSplitter** component and have it split the message according to XPath expression *"/orders/order"*. Create a route between the **cFile** and **cSplitter** components.
   c. Place a **cSetHeader** right after the **cSplitter** component. Link the **Split** route from the **cSplitter** component to the **cSetHeader** component.
   d. In the **cSetHeader** component, add two header variables:

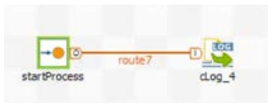
      "discount"; it gets its value from the **XPath** expression *"/order/discount/text()"*

      "orderID"; its value is evaluated by the **XPath** expression *"/order/@oID"*

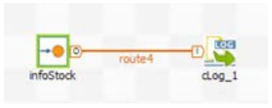
   e. Next to the **cSetHeader** component, add a **cBean** component.
   f. In the **Repository**, create a new bean and name it *processOrdersFile*.
   g. The code for this bean is stored in the file *processOrdersFile.txt* at *C:\StudentFiles\routes\r08_routingSlip*. Open it and inspect the code to understand what it does.
   h. Copy the code and paste it in the studio, in the new **processOrdersFile** bean (make sure this code replaces the bean skeleton generated when the bean was created). Save the bean and close its window.
   i. Configure the **cBean** component and select the **New Instance** option. In the **Bean class** parameter, enter *beans.processOrdersFile.class*. Select the **Specify the method** check box and enter *"defineRouteSequence"* in the text box.
   j. Create a route to link the **cSetHeader** component and **cBean** component.
   k. Add a **cRoutingSlip** component next to the **cBean** component and trace a route between them.
   l. In the **Component** view of the **cRoutingSlip**, specify the header variable in which the component will find the processing sequence, as well as the URI delimiter (the character used in the header variable to separate the different URIs).
      In the **Header name** text box, enter *"routeSequence"*. In the **URI delimiter**, keep the default *","*.



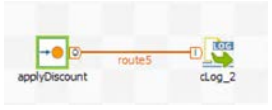
3. Create the startProcess route:
    a. Add a **cDirect** component and rename it *startProcess*. In the **Component** view, in the **Name** text box, enter *"startProcess"*
    b. Add a **cLog** component next to the **cDirect** component. In the **Logger Category**, enter *"MainLogger"*. Set the options to **Specify output log message** and enter *"Start processing Order #${in.header.orderID}"*.
    c. Create a route to link the **cDirect** component to the **cLog** component.



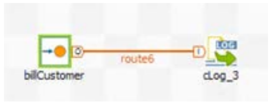
4. Create the infoStock route:
    a. Add a **cDirect** component and rename it *infoStock*. In the **Component** view, in the **Name** text box, enter *"infoStock"*
    b. Add a **cLog** component next to the **cDirect** component. In the **Logger Category**, enter *"stockLogger"*. Select the **Specify output log message** option and enter *"Send order information to the warehouse"*.
    c. Create a route to link the **cDirect** component to the **cLog** component.



5. Create the applyDiscount route:
    a. Add a **cDirect** component and name it *applyDiscount*. In the **Component** view, in the **Name** text box, enter *"applyDiscount"*
    b. Add a **cLog** component next to the **cDirect** component. In the **Logger Category**, enter *"discountLogger"*. Select the **Specify output log message** option and enter *"*** Apply customer discount ***"*
    c. Create a route to link the **cDirect** component to the **cLog** component.



6. Create the billCustomer route:
    a. Add a **cDirect** component and name it *billCustomer*. In the **Component** view, in the **Name** text box, enter *"billCustomer"*
    b. Add a **cLog** component next to the **cDirect** component. In the **Logger Category**, enter *"billLogger"*. Select the **Specify output log message** option and enter *"Charge the customer for order #${in.header.orderID}\n"*
    c. Create a route to link the **cDirect** component to the **cLog** component.



7. Run the route.
8. In a file explorer, navigate to *C:\StudentFiles\routes\r08_routingSlip*. Copy the file *orders.xml* and paste it in the **input** folder.
9. On the console, watch the log messages and make sure every item in the order is processed as expected.

Congratulations! You can move on to the next EIP.

Broadcasting a Message

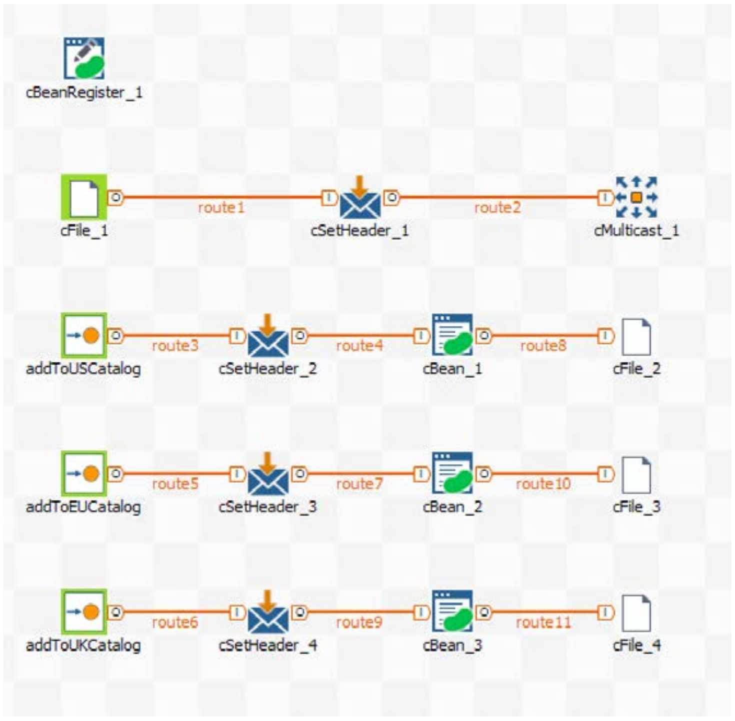
Overview

The Multicast EIP allows you to route the same message to a number of endpoints and process them in different ways.

In this lab, you receive an XML file with a new item to add to your product catalogs. You have three catalogs to update: one for the US, one for the UK, and one for the EU. Each has a different currency and ID to increment when adding a new product.

The process is not a sequence: there is no order or dependency between the catalogs. As a result, it is efficient to have the Multicast EIP simultaneously send the same message (with the product information) to three processing routes.

Each processing route handles integration of the new product in its catalog. One route does it for the US, another does it for the UK, and a third does it for the EU.



Before starting

1. In a file browser, go to *C:\StudentFiles\routes\r09_multicast* and open the file *manageCatalog.txt*. This is the bean class you use to fake the integration of the message in the catalog. Look at the code and make sure you understand what it does.
2. Copy the bean code.
3. Back in the studio, in the **Repository**, create a **new Bean** and name it *manageCatalog*.
4. Paste the code to replace any preexisting code in your new bean class. Save and close.

Implementing the cMulticast EIP

1. Create a new Camel context and name it *r09_multicast*.
2. Add a **cBeanRegister** component.
3. To create a new bean reference, in the **Component** view of the **cBeanRegister** component, enter *"myCatalogBean"* as the **Id**. In the **Class name** text box, enter *beans.manageCatalog*

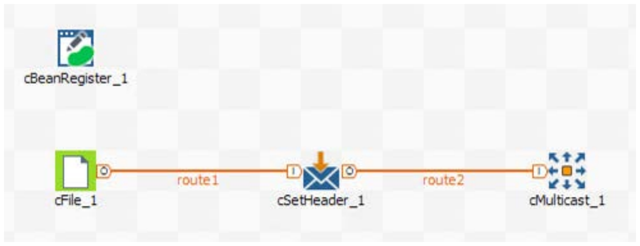
4. Create the first route:
    a. Add a **cFile** component to the design area and configure it to scan the following directory: *C:\StudentFiles\routes\r09_multicast\input*
    b. To extract data from the message, place a **cSetHeader** component to the right of the **cFile** component. Create three header variables as follows:

        Name: *"productID"* / **Language:** *XPath* / **Value:** *"/movie/@ASIN"*

        Name: *"price"* / **Language**: *XPath* / **Value:** *"/movie/price/text()"*

        Name: *"title"* / **Language:** *XPath* / **Value:** *"/movie/title/text()"*

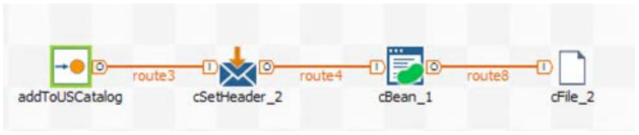
    c. Create a route from the **cFile** component to the **cSetHeader** component.
    d. Add a **cMulticast** component. You will configure it later.
    e. Create a route from the **cSetHeader** component to the **cMulticast** component.



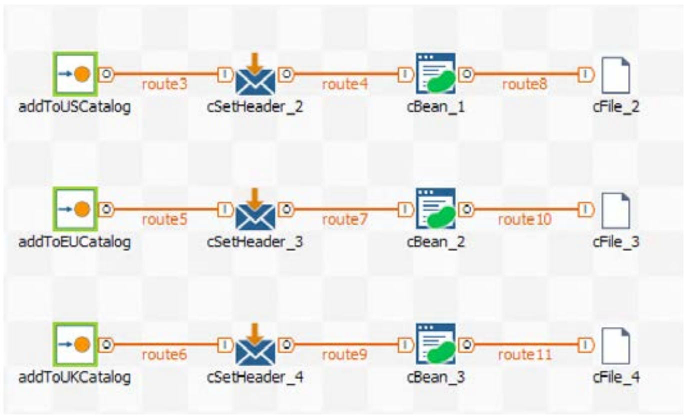
5. Create the second route, which will mock adding the product to the US catalog:
    a. Add a **cDirect** component and name it *addToUSCatalog*. In the **Component** view, set the **Name** parameter to *"addToUSCatalog"*
    b. Add a **cSetHeader** component next to the **cDirect** component. This component declares a new header variable named *countryCode*, which is a mandatory parameter of the bean method. Set the new variable as follows:

        Name: *"countryCode"* / **Language:** *Constant* / **Value:** *"US"*

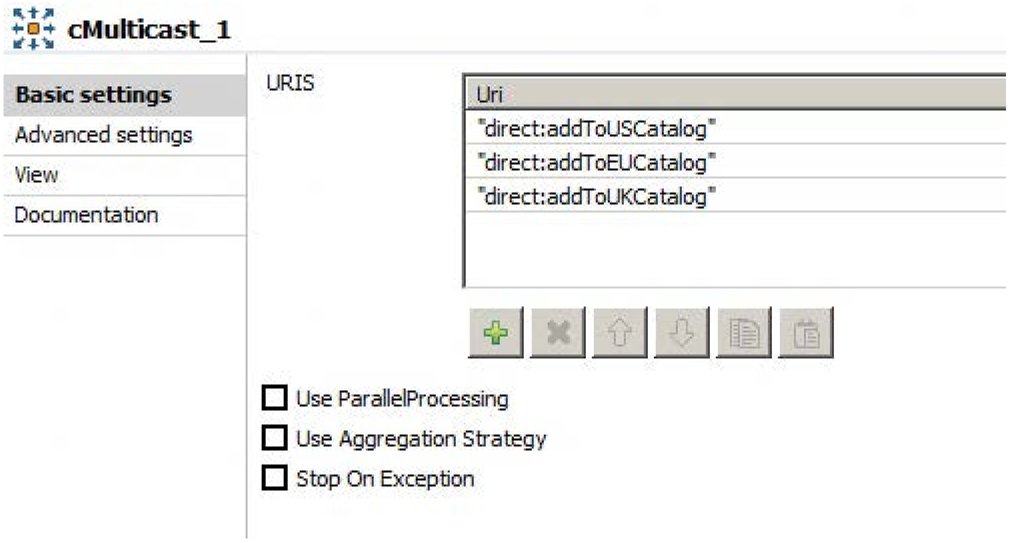
    c. Create a route from the **cDirect** component to the **cSetHeader** component.
    d. To process your message, add a **cBean** component next to the **cSetHeader** component and link them via a route.
    e. In the **Component** view of the **cBean** component, to call the reference you created in your **cBeanRegister**, select the **Reference** radio button. In the **Id** text box, enter *"myCatalogBean"*. Select the **Specify the method** check box and enter *"addNewProduct"* in the text box.
    f. Finally, add a **cFile** component and link it to the **cBean** component via a route.
    g. To configure the **cFile** producer component, in the **Path** text box, enter *"C:/StudentFiles/routes/r09_multicast/output/US"*. In the **fileName** text box, enter *"${in.header.title}.xml"*



6. To create the two remaining routes, **addToUKCatalog** and **addToEUCatalog**, repeat Step 5 by replacing any occurrences of US with UK, then EU.



7. Click the **cMulticast** component to configure it. The basic configuration takes a list of URIs to which the component must send its messages. To send the message to a route, provide the URI of the first component (the consumer) of a destination route. Click the **Plus (+)** sign to add a new destination URI and enter *"direct:addToUSCatalog"*. The message will be forwarded by the cMulticast component to the corresponding cDirect consumer component.
8. To send the same message to the two other destination routes, click the **Plus (+)** sign and add two other URIs: *"direct:addToEUCatalog"* and *"direct:addToUKCatalog"*.

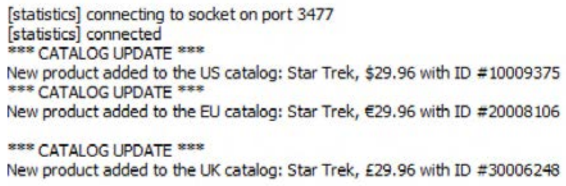


URIS

Uri
"direct:addToUSCatalog"
"direct:addToEUCatalog"
"direct:addToUKCatalog"

☐ Use ParallelProcessing
☐ Use Aggregation Strategy
☐ Stop On Exception

9. Run the Camel context.
10. In a file browser, go to *C:\StudentFiles\routes\r09_multicast* and copy the *movie6.xml* file to the **input** folder.
11. Inspect the **output** folders and log messages to make sure your file was processed as expected.



```
[statistics] connecting to socket on port 3477
[statistics] connected
*** CATALOG UPDATE ***
New product added to the US catalog: Star Trek, $29.96 with ID #10009375
*** CATALOG UPDATE ***
New product added to the EU catalog: Star Trek, €29.96 with ID #20008106

*** CATALOG UPDATE ***
New product added to the UK catalog: Star Trek, £29.96 with ID #30006248
```

You have completed the last lab in this chapter. It is time to wrap up.

Wrap-up

In this chapter you learned more about EIPs and how to use them in the context of ESB development.

You can now integrate applications and process messages with four of the most used EIPs:

- cMessageRoute, which routes messages by applying a condition to their content
- cSplitter, which splits a message into several smaller ones based on an expression
- cRoutingSlip, which guides a message through a sequence of processing routes defined in a header variable
- cMulticast, which simultaneously sends a message to multiple destinations

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let’s continue >**.

LESSON 6

Calling DI Jobs from Routes

This chapter discusses the following.

Overview

An ESB route can interact with a Data Integration (DI) Job to process data and pass information back to the running route. In this lab, you receive an XML file with a collection of items. These items can be either book or movie references, stored in a very generic format. You need to process the data and change the XML structure of these items, specifying it for movies or books.

This process requires data processing capabilities, so you will call DI Jobs to read the data and map it to another XML structure.

Objectives

After completing this lesson, you will be able to:

Design DI Jobs using specific route input and output components

Design and run a route that calls a DI Job

Design and run a route that takes data from the DI Job and uses it for output

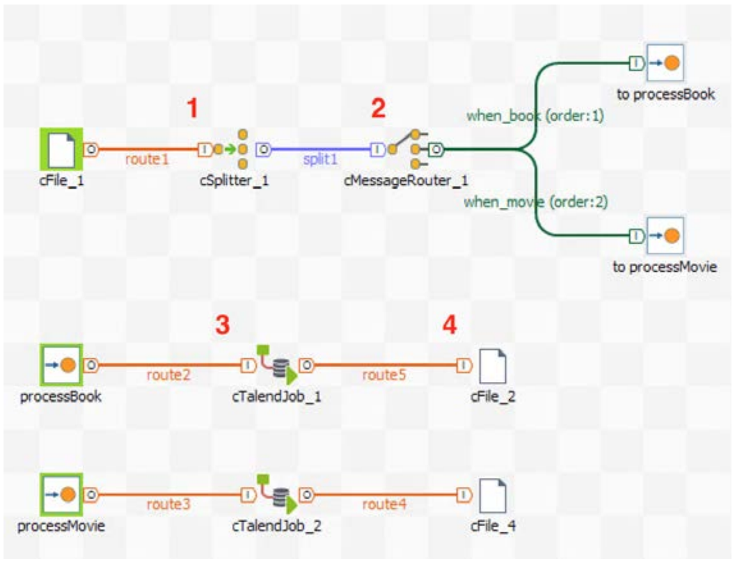
First you will design the route that receives the XML message.

Creating a Route to Call DI Jobs

Overview

In this lab, the route:

1. Splits the original message into smaller messages
2. Routes the message to a destination according to its content
3. Passes the message to a DI Job via the cTalendJob component
4. Gets the message back from the DI Job and produces it in a file



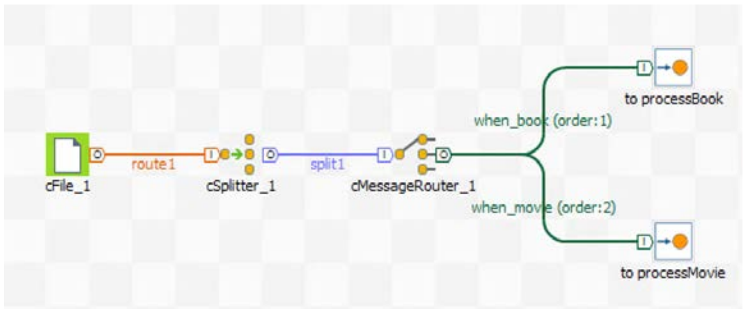
Note: Book and movie items are sent to different routes and different DI jobs.

Creating the route

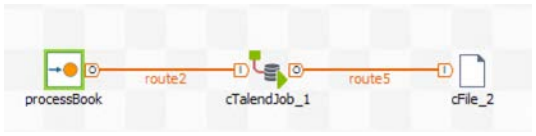
1. Create a new Camel context and name it *r10_callJobs*.
2. Build the first route:
   a. Add a **cFile** component and configure it to consume in the folder *"C:\StudentFiles\routes\r10_callJobs\input"*
   b. Add a **cSplitter** component and use a route to link it with the **cFile** component.
   c. Configure the **cSplitter** component using the following parameters:

      Language: *XPath* / **Expression:** *"/catalog/item"*

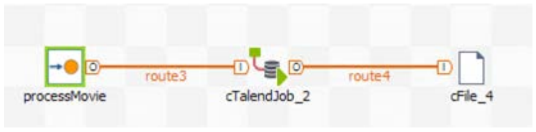
   d. Add a **cMessageRouter** component next to the **cSplitter** component. Link the **Split** route to the **cMessageRouter** component.
   e. Add two **cDirect** components. Name them *to processBooks* and *to processMovie*
   f. From the **cMessageRouter** component, drag two **When** triggers and link them to both **cDirect** components.
   g. Rename the **When** link that connects to the **processBooks cDirect** component *when_book*. Click the **When** link, and, in the **Component** view, configure the **Type** as **xpath** and enter *"/item/@type = 'book'"* in the **Expression** text box.
   h. Rename the **When** link that connects to the **processMovies cDirect** component *when_movie*. Click the **When** link, and, in the **Component** view, configure the **Type** as **xpath** and enter *"/item/@type = 'movie'"* in the **Expression** text box.



3. Create the second route, which will process the book items:
   a. Add a **cDirect** component, name it *processBook*, and enter *"processBook"* as the **Name**.
   b. Add a new **cTalendJob** component next to the **cDirect** component.
   c. Trace a route from the **cDirect** component to the **cTalendJob** component.
   d. Add a **cFile** component after the **cTalendJob** component. In the **Component** view, in the **Path** parameter, enter *"C:/StudentFiles/routes/r10_callJobs/output/books/"*. In the **fileName** text box, enter *"${in.header.title}.xml"*
   e. Use a route to link the **cTalendJob** component to the **cFile** component.



4. Create the third route, which will process the movie items:
   a. Add a **cDirect** component, name it *processMovie*, and enter *"processMovie"* as the **Name**.
   b. Add a new **cTalendJob** component next to the **cDirect** component.
   c. Trace a route from the **cDirect** component to the **cTalendJob** component.
   d. Add a **cFile** component after the **cTalendJob** component. In the **Component** view, in the **Path** parameter, enter *"C:/StudentFiles/routes/r10_callJobs/output/movies/"*. In the **fileName** text box, enter *"${in.-header.title}.xml"*
   e. Use a route to link the **cTalendJob** component to the **cFile** component.



You will now create DI Jobs to process the messages.

Creating DI Jobs with Route Components

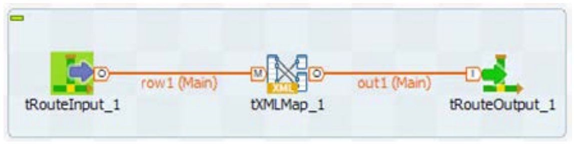
Overview

To be eligible for route calls, DI Jobs must start and end with specific components: tRouteInput and tRouteOutput. With these two components, you can create any DI process you want.

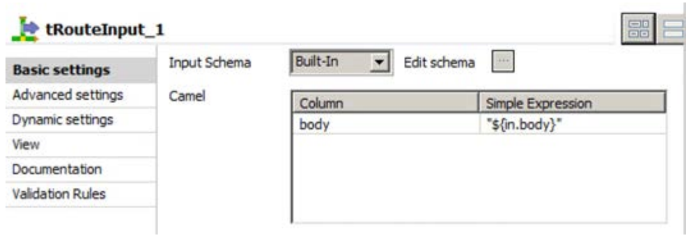
Create the first route-compatible Job

1. In the **Repository**, create a new DI Job and name it **processBooks**.
2. In the design area, add the **tRouteInput**, **tXMLMap** and **tRouteOutput** components.
3. Link the components via a **Main** flow, in this order:

   tRouteInput → tXMLMap → tRouteOutput



4. Select the **tRouteInput** component and go to the **Component** view. The tRouteInput default configuration retrieves the body of the message by using a Simple expression. Remember that you can retrieve more than just the body by adding more columns in the schema and assigning them Simple expressions.



Input Schema: Built-In

Camel

Column	Simple Expression
body	"${in.body}"

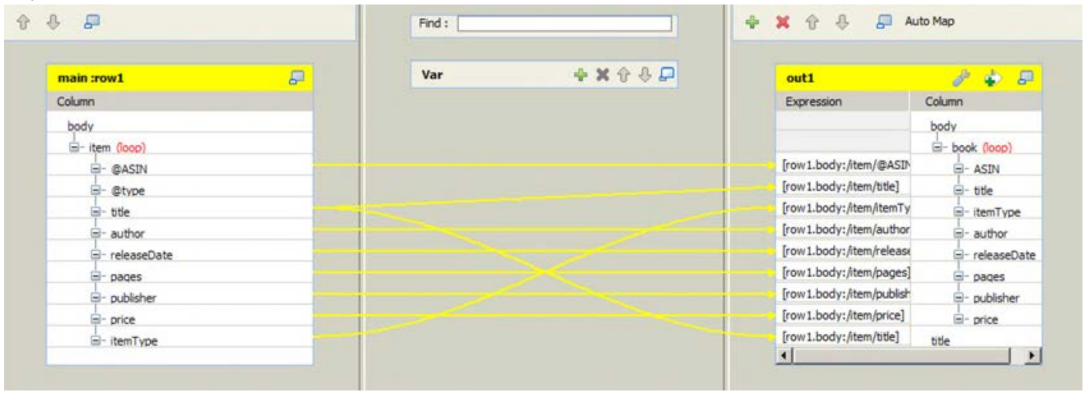
5. Open the **tXMLMap** component.
6. In the **input** row of the **tXMLMap**, right-click the **body** element, and, on the menu, choose **Import From File**. The schema of the book item is stored in a file at the following location:
   C:\StudentFiles\routes\r10_callJobs\schemas\original_book_schema.xml
7. In the **tXMLMap** output, create two new columns:

   Column: *body* / **Type:** *Document*

   Column: *title* / **Type:** *String*

8. In the **tXMLMap** output, right-click the **body** element. On the menu, choose the **Import From File** option. The schema you need is stored in the following location:
   C:\StudentFiles\routes\r10_callJobs\schemas\output_book_schema.xml

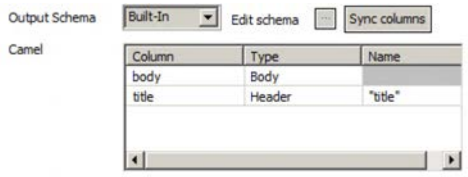
9. Map the data as shown in the screenshot below:



10. To exit the **tXMLMap** interface, click **OK**.
11. Click the **tRouteOutput** component and go to its **Component** view. This is where you can define which information to pass to the route. Click the **Edit Schema ellipsis (...)** button.
12. The **Column** parameter designates the columns of data coming from your Job (in this case, from the tXMLMap component).
    The **Type** parameter indicates whether the column will be the body, the header, or a property of your exchange and message. Therefore, a column can pass to the route as the body of a message, a header, a property, or system information.
    The **Name** parameter applies only to headers and properties. You can name the header or property that you want to pass to the route.
    Set the configuration of the **tRouteOutput** as follows:

    Column: *body* / **Type:** *Body*

    Column: *title* / **Type:** *Header* / **Name**: *"title"*



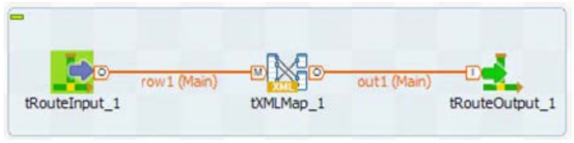
Column	Type	Name
body	Body	
title	Header	"title"

13. Save the Job.

Create the second Job

This second job dealing with movie items is similar to the first. These instructions are less detailed; if needed, refer to the instructions for the first route.

1. In the **Repository**, create a new DI Job and name it **processMovies**.
2. Add the components **tRouteInput**, **tXMLMap** and **tRouteOutput** to the design area and link them as shown in the screenshot below:



3. Open the **tXMLMap** component.
4. Import the schema of the input **body** column from the file *C:\StudentFiles\routes\r10_callJobs\schemas\original_movie_schema.xml*

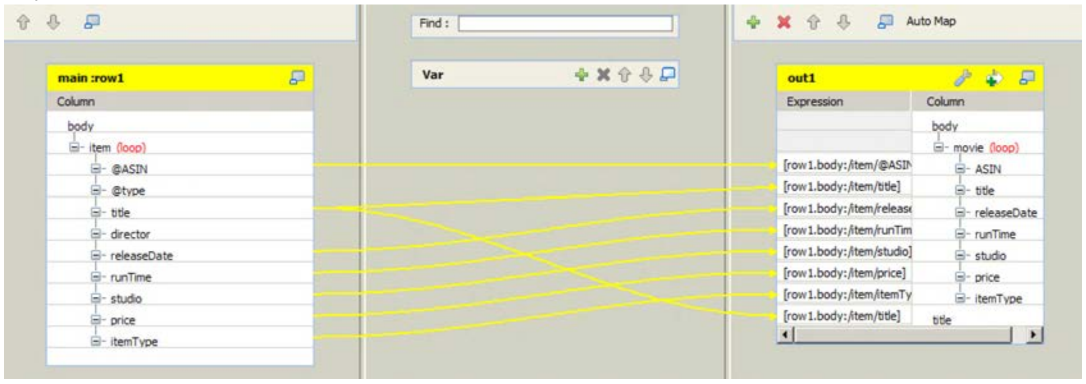
5. In the **output** flow, create two columns:

   Column: *body* / **Type:** *Document*

   Column: *title* / **Type:** *String*

6. Import the schema of the output **body** column from the file *C:\StudentFiles\routes\r10_callJobs\schemas\output_movie_schema.xml*

7. Map the data as shown in the screenshot below:



Expression	Column
	body
	movie (loop)
[row1.body:/item/@ASIN	ASIN
[row1.body:/item/title]	title
[row1.body:/item/release	releaseDate
[row1.body:/item/runTim	runTime
[row1.body:/item/studio]	studio
[row1.body:/item/price]	price
[row1.body:/item/itemTy	itemType
[row1.body:/item/title]	title

8. In the **tRouteOutput** component, set the following configuration:

   Column: *body* / **Type:** *Body*

   Column: *title* / **Type:** *Header* / **Name:** *"title"*

9. Save the Job.

These DI Jobs cannot be executed on their own; they need to be called by a route. We will go back to the route to finalize configuration.

Configuring the cTalendJob Component

Overview

The last step in this lab is to connect your Camel context to your DI Jobs. This communication is handled by the cTalendJob component.

Completing configuration

1. In the **r10_callJobs** route, in the **processBook** route, select the **cTalendJob** component.
2. Go to the **Component** view. The configuration is very similar to that of the tRunJob component in DI. Keep the default **Repository** option selected, as well as the **Use Selected Context** option.
3. Click the **Ellipsis [...]** button of the **Job** parameter. A new **Assign Job** window opens.
4. You are prompted to either create a new Job or use an existing one. Choose the **Assign an existing Job to this cTalendJob component** option and click **Next**.
5. In the **Repository**, select the **processBooks** Job and click **Finish**.
6. In the **processMovie** route, select the **cTalendJob** component and configure it to run the **processMovies** Job.
7. Save the route.

Running the route

1. Run the route.
2. In a file browser, navigate to *C:\StudentFiles\routes\r10_callJobs*. Copy the *newCollection.xml* file and paste it in the **input** folder.
3. When the file is processed, you should find movie and book items stored in the appropriate folders in the output folder. Each item should be saved with its title as the file name, with the file structure defined in the tXMLMap components.

Next step

You have almost completed this lesson. Continue to the Wrap-up section for a review of the concepts we covered.

Wrap-up

You learned how to create a route that calls Jobs. This allows you to benefit from the enhanced data processing abilities of DI Jobs while running mediation routes and processing messages in real time.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let’s continue >**.

LESSON 7

Exploring More EIPs

This chapter discusses the following.

Overview

This chapter introduces you to three more new EIPs that will let you better handle and process messages.

Objectives

After completing this lesson, you will be able to:

- Use a non-intrusive Wire Tap EIP to catch messages and duplicate them to a parallel process
- Load balance a process between two or more routes
- Regulate traffic within a route by using the Throttler EIP

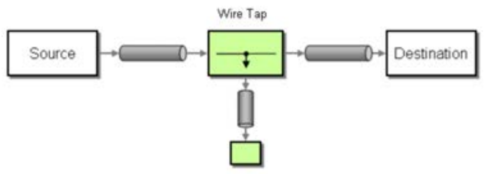
Next step

When you are ready, move on to the first lab and learn about the cWireTap EIP.

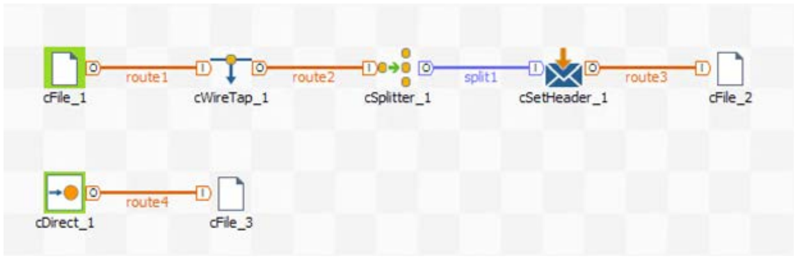
Duplicating a Message

Overview

The Wire Tap EIP is used to duplicate a message in the exact state it is in when it reaches the EIP. This EIP does not block or disrupt the process in any way, and as soon as the message passes through the component, it is copied and sent to both the next component in the route and a second destination set as a parameter.

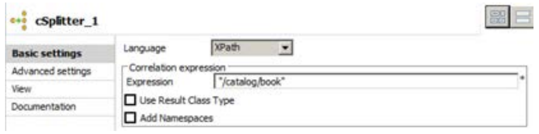


In this very simple lab, a message is intercepted and duplicated by the cWireTap component (the Talend implementation of the Wire Tap EIP). The original message goes on to the next component (cSplitter), while a copy of the message is sent by the cWireTap component to another route for logging purposes.



Using the cWireTap component

1. Create a new Camel context and name it *r11_wireTap*.
2. Create the first route:
    a. Add a **cFile** component to the design area and configure it to scan the following directory: *"C:/StudentFiles/routes/r11_wireTap/input"*
    b. Add a **cWireTap** component next to the **cFile** component. As you can see in its **Component** view, the main parameter is the destination **URI** for the copy of the message. You will configure this URI later.
    c. Add a **cSplitter** component to the right of the **cWireTap** component. Open the file at *C:\StudentFiles\routes\r11_wireTap\books.xml* and deduce the splitter **XPath** query. Configure the **cSplitter** component. If you need help, see the screenshot below.



Language: XPath

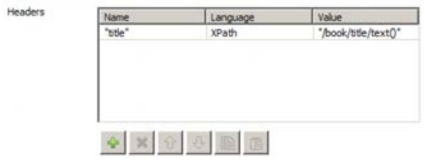
Correlation expression

Expression: "/catalog/book"

☐ Use Result Class Type

☐ Add Namespaces

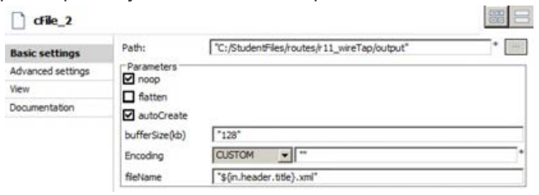
    d. Add a **cSetHeader** component to the right of the **cSplitter** component. Configure it to create a new header variable named *"title"* that retrieves the title tag from the XML message. If you need help, see the screenshot below.



Headers

Name	Language	Value
"title"	XPath	"/book/title/text()"

e. Finally, add a **cFile** component and configure it to produce its files in the *"C:/StudentFiles/routes/r11_wireTap/output"* repository. Set the **fileName** parameter so that files are named after the book titles in the messages.



cFile_2

Basic settings | Advanced settings | View | Documentation

Path: "C:/StudentFiles/routes/r11_wireTap/output"

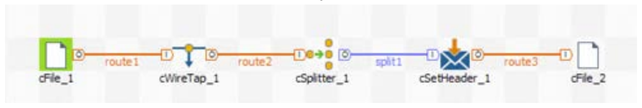
Parameters
- ☑ noop
- ☐ flatten
- ☑ autoCreate

bufferSize(kb): "128"

Encoding: CUSTOM ""

fileName: "${in.header.title}.xml"

f. Create the route between the components as shown below:

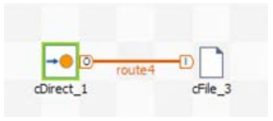


3. Create the second route:

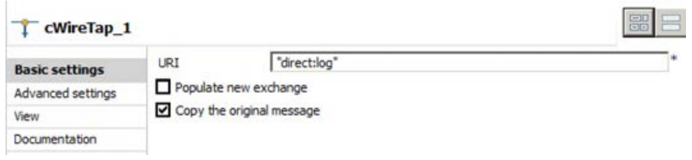
   a. Add a **cDirect** component to the design area. In the **Component** view, in the **Name** text box, enter *"log"*

   b. Add a **cFile** component next to the **cDirect** component. Configure it to produce files in the *"C:/StudentFiles/routes/r11_wireTap/log" folder*. In the **fileName** text box, enter *"catalog-${date:now:yyyyMMddhhmmss}.xml"*

   c. Draw a route from the **cDirect** component to the **cFile** component.



4. To configure the Wire Tap EIP, click the **cWireTap** component. In its **Component** view, in the **URI** text box, enter the URI of your **cDirect** component.



cWireTap_1

Basic settings | Advanced settings | View | Documentation

URI: "direct:log"

- ☐ Populate new exchange
- ☑ Copy the original message

5. Run the Camel context.

6. In a file browser, go to *C:\StudentFiles\routes\r11_wireTap* and copy the *books.xml* file. Paste it in the **input** folder.

7. Inspect the **output** and log folders. The same message was sent to two processes, one of which produced multiple files in the output folder, whereas the second one produced files in the log folder.

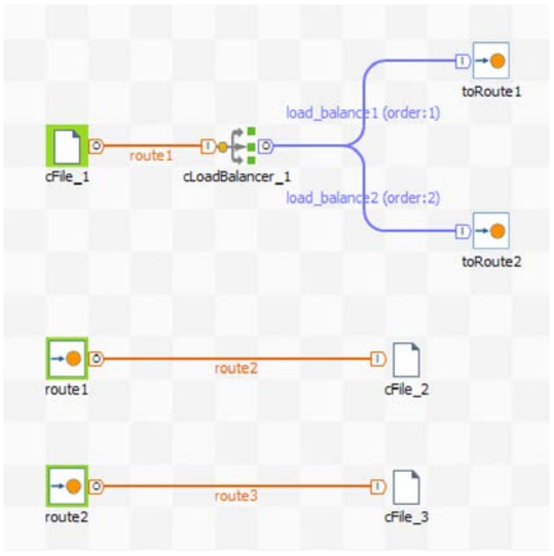
You have completed this lab and can move on to the next EIP.

Load Balancing

Overview

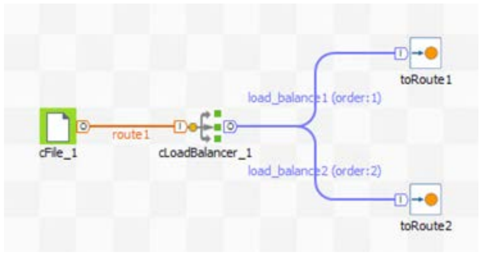
The Load Balancer EIP allows you to delegate messages to one of a number of endpoints using a variety of different load-balancing strategies.

In this lab, the Camel context must consume several files and use the cLoadBalancer EIP to evenly distribute the messages to two destination routes.



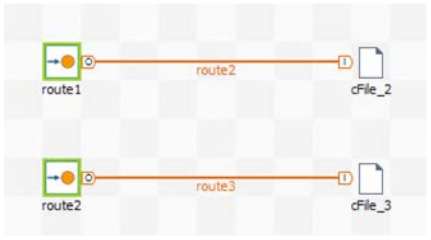
Using the cLoadBalancer component

1. Create a new Camel context and name it *r12_loadBalance*.
2. Create the first route:
    a. Add a **cFile** component to the design area and configure it to scan the *"C:/StudentFiles/routes/r12_loadBalance/input"* directory.
    b. Add a **cLoadBalancer** component to the right of the **cFile** component and draw a route between them.
    c. Add two **cDirect** components to the right of the **cLoadBalancer** component. Name them *toRoute1* and *toRoute2*.
    d. Right-click the **cLoadBalancer** component and choose **Row > Load Balance**. Draw a link from the **cLoadBalancer** component to the **cDirect toRoute1** component. Do the same with the **cDirect toRoute2** component.



    e. Click the **cLoadBalancer** component and go to the **Component** view. Click the **Strategy** menu and inspect the available strategies.
    f. Select the **Round Robin** strategy. This is similar to the Round-robin algorithm employed by processes and network schedulers: each destination is given an equal share of messages in turn.

3. Create the destination routes:
   a. Add a **cDirect** component to the design area and name it *route1*. In the **Component** view, in the **Name** text box, enter *"route1"*
   b. Add a **cFile** component to the right of the **cDirect** component. Configure it to produce files in the *"C:/StudentFiles/routes/r12_loadBalance/output/route1"* folder.
   c. Draw a route from the **cDirect** component to the **cFile** component.
   d. Create a third route by repeating steps a, b, and c, but replace all occurrences of *route1* with *route2*.



4. Configure the **cDirect** producer components from the first route so that the **cDirect toRoute1** component points to the **cDirect route1** consumer, and the **cDirect toRoute2** component points to the **cDirect route2** consumer.
5. Run the Camel context.
6. In a file browser, go to *C:\StudentFiles\routes\r12_loadBalance* and copy the *movieX.xml* files. Paste them in the **input** folder.
7. Inspect the **output** folders and make sure the files were distributed evenly and processed by both *route1* and *route2*

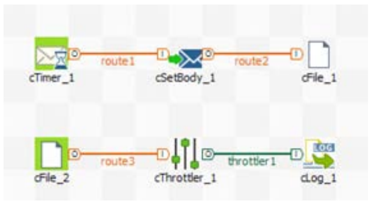
Good job! You can move on to the last EIP lab.

Throttling and Delaying Messages

Overview

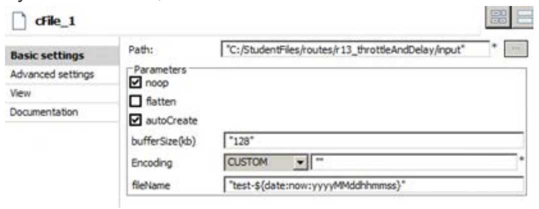
The Throttler EIP allows you to ensure that a specific endpoint does not get overloaded, or that you do not exceed a service-level agreement with an external service. The cThrottler component lets you define the maximum number of messages you allow to be processed in a certain amount of time. If the number of messages arriving in the route exceeds the number allowed, new messages are queued and delayed.

In this lab, the first route creates one file per second. The second route is in charge of consuming and processing the files. However, the cThrottler component authorizes only one message to be processed in a period of 4 seconds.



Using the cThrottler component

1. Create a new Camel context and name it *r13_throttleAndDelay*.
2. Create the first route:
   a. Add a **cTimer** component to the design area and leave the default configuration.
   b. Add a **cSetBody** component to the right of the **cTimer** component. Feel free to add content to the body of the message.
   c. Add a **cFile**component to the right of the **cSetBody** component. Configure it to produce files in the *"C:/StudentFiles/routes/r13_throttleAndDelay/input"* folder.
   Set the **fileName** parameter so that each file is created with a unique name.
   If you need a hint, see the screenshot below.



cFile_1

Basic settings | Advanced settings | View | Documentation

Path: "C:/StudentFiles/routes/r13_throttleAndDelay/input"

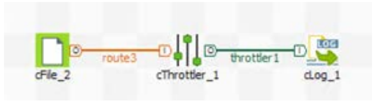
Parameters
- ☑ noop
- ☐ flatten
- ☑ autoCreate

bufferSize(kb): "128"

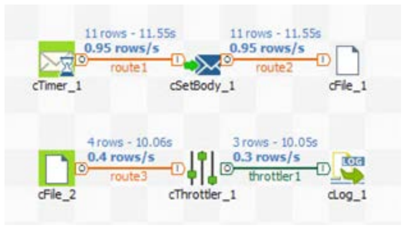
Encoding: CUSTOM ""

fileName: "test-${date:now:yyyyMMddhhmmss}"

3. Create the second route:
   a. Add a **cFile** component to the design area. In the **Component** view, in the **Path** text box, enter *"C:/StudentFiles/routes/r13_throttleAndDelay/input"*. With this parameter set, the **cFile** component consumes the files produced by the first route.
   b. Add a **cThrottler** component and use a route to connect it to the **cFile** component.
   c. In the **Component** view of the **cThrottler** component, set the number of **Requests per period** to *1*. Select the **Set time period** check box and enter *4000* in the text box. Now the **cThrottler** component will only accept one request per 4-second period in the route.
   d. Add a **cLog** component and use a route to connect it to the **cThrottler** component.
   e. In the **cLog Component** view, select the **Specify output log message** option and enter *"file processed"*



4. Run the route.

5. Inspect the *"C:/StudentFiles/routes/r13_throttleAndDelay/input"* folder. You will see some files being produced and not being consumed right away.
   In the **Studio**, see the statistics showing the difference between the number of files produced and the number consumed by the throttler route.
   

Next step

You have almost completed this lesson. Continue to the Wrap-up section for a review of the concepts we covered.

Wrap-up

In this chapter, you learned more about more-advanced EIPs. You can duplicate a message on the fly, without blocking or disrupting its original route, thanks to the cWireTap EIP.
You also know how to relieve an overloaded route by distributing messages to several destinations with the cLoadBalancer EIP.
Finally, you configured the cThrottler EIP to prevent a route from being overloaded with many messages at the same time by delaying some of them.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

LESSON 8

Messaging with a Dedicated Broker

This chapter discusses the following.

Overview

As shown in the slides, message brokers such as ActiveMQ have a sole purpose: making sure messages are delivered to their recipients. In the world of ESB design and infrastructure, using a message broker is a best practice, as it ensures that a message leaving a system reaches its destination. Message brokers handle all transfer complexity and possible communication errors.

In this lab, two routes communicate:

The first route receives a message, processes it, and sends it to a message queue on the broker ActiveMQ.

The second route consumes any message appearing on this same queue, and processes it as well.

Objectives

After completing this lesson, you will be able to:

Connect a route to a message broker

Use the cJMS component as a producer to standardize a message and send it to a message queue

Use the cJMS component as a consumer to read a message from a message queue

Exchange messages between two routes

Confirm that the message broker can handle errors and prevent message loss

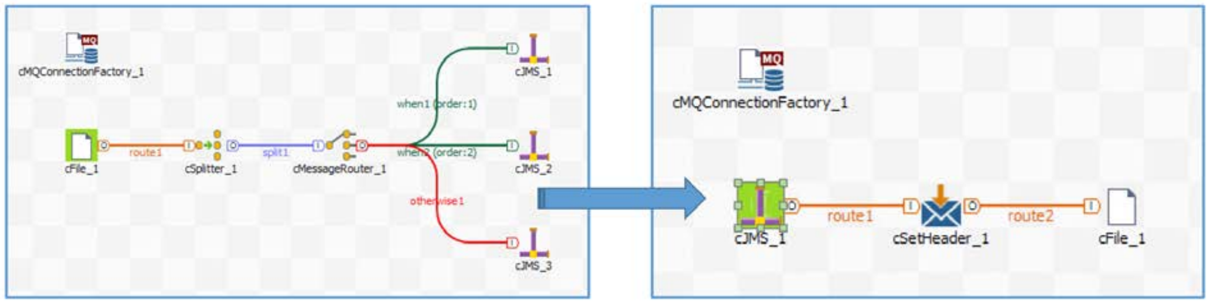
Next step

When you are ready, move on to the lab instructions.

Sending Messages to ActiveMQ

Overview

In this lab, two different Camel contexts exchange messages via the message broker, ActiveMQ.



This lab introduces you to two new components: cMQConnectionFactory and cJMS.

The cMQConnectionFactory component, like the database connection components in DI Jobs, initiates a connection to the message broker connection pool. It keeps the connection open, as well as handling connection exceptions and network issues.

The cJMS component uses the cMQConnectionFactory connection to exchange data with the message broker. When used as a consumer endpoint, the cJMS component connects to a queue or a topic on the broker and consumes any messages stored there. When used as a producer, the cJMS component sends its message to a queue or a topic on the message broker.

As you can see in the screenshot above, the first Camel context sends messages to a queue on the broker by using cJMS components, while the second Camel context connects to the same queue via a cJMS consumer component and consumes the messages.

Starting the message broker

Before developing, you need to make sure that the message broker ActiveMQ is up and running.

1. In a file explorer, browse to *C:\Talend\6.2.1\esb\activemq\bin\win64\*
2. To start the ActiveMQ server, double-click **activemq.bat**.
3. As you can see from the logs in the console window, the ActiveMQ broker interface starts on *tcp://127.0.0.1:61616* by default. You may also notice that it starts an interface on localhost (*127.0.0.1*) on port 8161. This is the administration interface.
4. Open an Internet browser and connect to the ActiveMQ administration URL, *http://localhost:8161/admin*
5. The interface administration page requires a log-in ID and password. By default, these are *admin* and *admin*
6. This interface allows developers to test and debug routes or DI Jobs when producing and consuming on the message broker. The Queues and Topics sections make it very easy to inspect the available queues and topics, as well as their messages. Keep this page open so you can use it later to confirm that your first Camel context has successfully produced its messages.

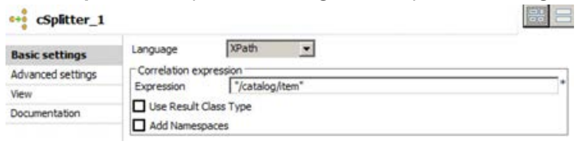
Connecting to the message broker

1. Create a new Camel context and name it *r14_sendJMS*.
2. Add a **cMQConnectionFactory** component to the design area.
3. Go to the **Component** view of the **cMQConnectionFactory** component.
4. In the **MQServer** parameter, make sure you are connecting to an ActiveMQ server. As you can see in the drop-down menu, it is possible to connect to several other brokers as well.
5. In the **Broker URI** text box, enter *"tcp://localhost:61616"*

Creating the first route

In this route, you need to read a file that contains product information for a catalog of DVDs and Blu-ray discs. Each movie item has a language tag describing the soundtracks available on each disc. The tag can be languages "EN", "FR", or a combination. Take a quick look at the file at *C:\StudentFiles\routes\r14_sendJMS\catalog.xml*.

1. Add a **cFile** component and configure it to consume from the folder *C:\StudentFiles\routes\r14_sendJMS\*
2. To break the original message into several smaller ones each holding one movie item, to the right of the **cFile** component, add a **cSplitter** component. Configure it to split the message for each item. If you need help, see the screenshot below.

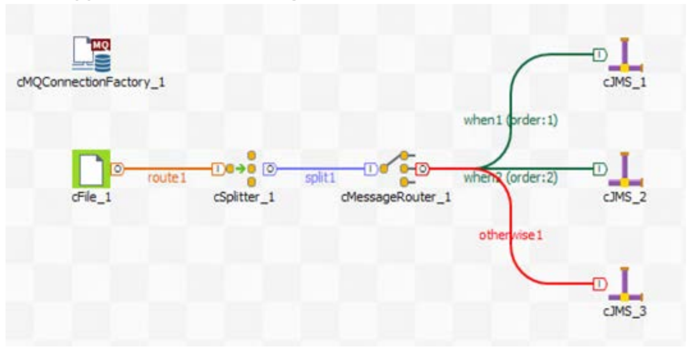


cSplitter_1

Basic settings | Advanced settings | View | Documentation

Language: XPath
Correlation expression
Expression: "/catalog/item"
☐ Use Result Class Type
☐ Add Namespaces

3. Draw a route from the **cFile** component to the **cSplitter** component.
4. To route the messages according to language tag, add a **cMessageRouter** component next to the **cSplitter** component.
5. Use the **Split** row of the **cSplitter** component and link it to the **cMessageRouter** component.
6. After the **cMessageRouter** component, add three **cJMS** components.
7. From the **cMessageRouter** component, use the **When** trigger to connect to two of the **cJMS** components. Use the **Otherwise** trigger to link the **cMessageRouter** component to the third **cJMS** component.

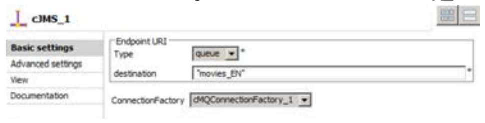


8. Select the **cJMS_1** component and go to its **Component** view. Set the parameters as follows:

   Type: *Queue*

   destination: *"movies_EN"*

   ConnectionFactory: *cMQConnectionFactory_1*



cJMS_1

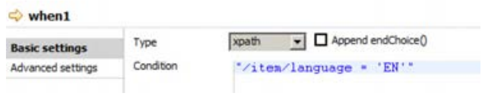
Basic settings | Advanced settings | View | Documentation

Endpoint URI
Type: queue
destination: "movies_EN"
ConnectionFactory: cMQConnectionFactory_1

9. Configure the **When1** link (connected to the **cJMS_1** component) and set the condition like this:

   Type: *xpath*

   Condition: *"/item/language = 'EN'"*



when1

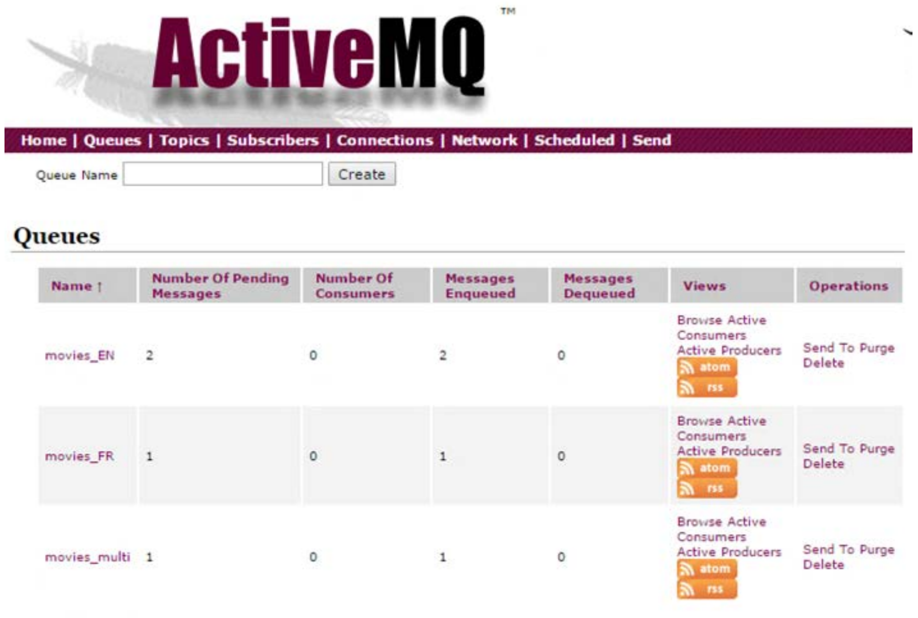
Basic settings | Advanced settings

Type: xpath ☐ Append endChoice()
Condition: "/item/language = 'EN'"

10. Select the **cJMS_2** component and set its parameters to produce messages in a queue named *"movies_FR"*
11. Configure the **When2** link and set its *xpath* condition to *"/item/language = 'FR'"*
12. Select the **cJMS_3** component and set its parameters so that it produces messages in a queue named *"movies_multi"*

Running the first Camel context

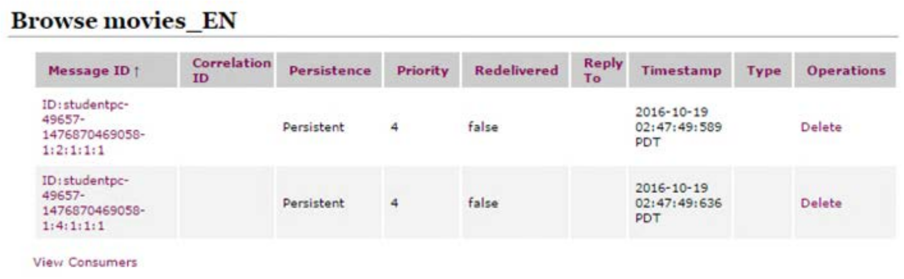
1. Run the route.
2. In a file explorer, browse to *C:\StudentFiles\routes\r14_sendJMS*
3. Copy the file *catalog.xml* and paste it in the **input** folder.
4. In the Studio, make sure the file is consumed and the messages are sent to the destinations.
5. Because no component is configured to consume the messages, they are stored in their queues until you build and run the second Camel context. This gives you the opportunity to check the ActiveMQ administration interface. Open an Internet browser and connect to *http://localhost:8161/admin*
6. Click the **Queues** link. The Queues page displays the available queues, along with details such as the number of messages enqueued and dequeued and number of consumers connected to the queue.


Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send
Queue Name
Create

Queues

Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
movies_EN	2	0	2	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete
movies_FR	1	0	1	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete
movies_multi	1	0	1	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete

7. Click one of the queues and drill down to the message level. For the queue you selected, you can see all pending messages.

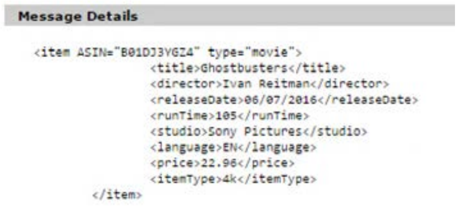


Browse movies_EN

Message ID ↑	Correlation ID	Persistence	Priority	Redelivered	Reply To	Timestamp	Type	Operations
ID:studentpc-49657-1476870469058-1:2:1:1:1		Persistent	4	false		2016-10-19 02:47:49:589 PDT		Delete
ID:studentpc-49657-1476870469058-1:4:1:1:1		Persistent	4	false		2016-10-19 02:47:49:636 PDT		Delete

View Consumers

8. Click one **Message ID** and inspect the content of the message. As you can see, Camel headers are passed via JMS in the JMS Properties. You can also read the body of the message.



Message Details

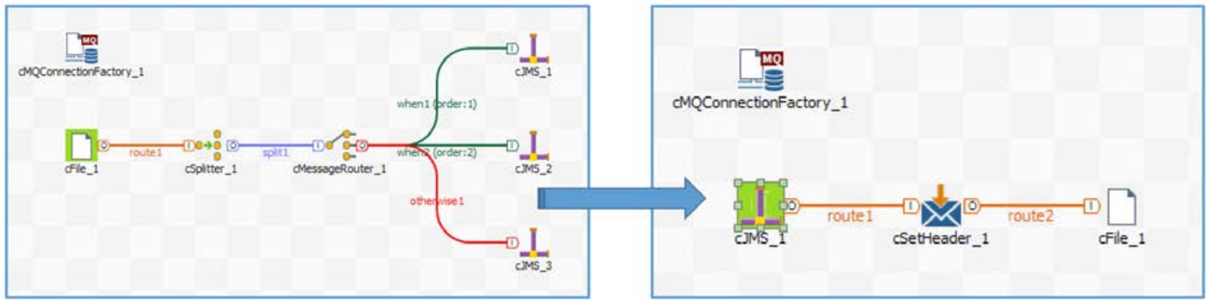
```
<item ASIN="B01DJ3VGZ4" type="movie">
                <title>Ghostbusters</title>
                <director>Ivan Reitman</director>
                <releaseDate>06/07/2016</releaseDate>
                <runTime>105</runTime>
                <studio>Sony Pictures</studio>
                <language>EN</language>
                <price>22.96</price>
                <itemType>4k</itemType>
        </item>
```

The first Camel context is ready, and you are now ready to develop the consumer context.

Reading Messages from ActiveMQ

Overview

This section shows you how to use a cJMS consumer endpoint.



This corresponds to the second Camel context (in the above screenshot on the right).

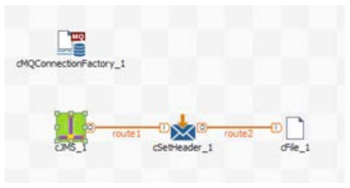
Connecting to the message broker

1. Create a new Camel context and name it *r14_readJMS*.
2. Add a **cMQConnectionFactory** component to the design area.
3. Go to the **Component** view of the **cMQConnectionFactory** component.
4. In the **MQServer** parameter, make sure you are connecting to an **ActiveMQ** server. As you can see in the drop-down menu, it is possible to connect to several other brokers as well.
5. In the **Broker URI** text box, enter *"tcp://localhost:61616"*

Creating the second route

In this route, you connect to one queue on the broker.

1. Add a **cJMS**, **cSetHeader**, and **cFile** component. Link them via routes as shown in the screenshot below.



2. To consume messages from the **movies_EN** queue, in the **Component** view of the **cJMS** component, set the following parameters:

   Type: *queue*

   destination: *"movies_EN"*

   Connection Factory: *cMQConnectionFactory_1*

3. In the **Component** view of the **cSetHeader** component, create a new header variable with the following parameters:

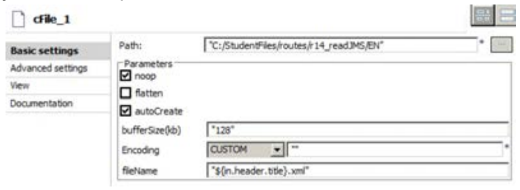
   Name: *"title"*

   Language: *XPath*

   Value: *"/item/title/text()"*

4. In the **Component** view of the **cFile** component, set the parameters so that the component produces files in the *C:\StudentFiles\routes\r14_readJMS\EN* folder, and each file is named after the title header variable plus the XML extension. If

you need help, see the screenshot below.



cFile_1

Basic settings

Advanced settings

View

Documentation

Path: "C:/StudentFiles/routes/r14_readJMS/EN"

Parameters

- [✓] noop
- [ ] flatten
- [✓] autoCreate

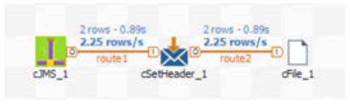
bufferSize(kb) "128"

Encoding CUSTOM ""

fileName "${in.header.title}.xml"

Consuming messages in the queue

1. Run the route. It should consume two messages.



2. In the Internet browser, on the **ActiveMQ administration** page, to see the available queues, click the **Queues** link.
3. In the list of queues, notice that the **movies_EN** queue has no more pending messages. The messages were consumed and subsequently removed from the queue.
4. In a file explorer, navigate to *C:\StudentFiles\routes\r14_readJMS\EN*. The messages that the second route consumed were produced as files here.

Next step

You have almost completed this lesson. Continue to the Wrap-up section for a review of the concepts we covered.

Wrap-up

In this lab, you learned how to connect to a message broker and how to produce and consume messages on it. You can now exchange messages using ActiveMQ, a fail-safe and robust module of the ESB infrastructure.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let’s continue >**.

LESSON 9

Developing SOAP Web Services

This chapter discusses the following.

Overview

SOAP web services use WSDL files. A WSDL file provides a contract between the service provider and the service consumer. It identifies the service operations and expected request/response format for each operation.

This lab will show you how to create both Web services and Web-service-consuming jobs. Designing a SOAP Web service in Talend Studio starts with design of the WSDL file. A specific GUI helps you design the required WSDL objects (for example, the service, binding, port type). Then, each service operation can be bound to a DI Job. Designing the operations of a Web service—that is, the part of the service that processes requests—is simple: operations are DI Jobs with specific input and output components.

The following labs show you how to create simple to more-complex Web services, and how to consume them.

Objectives

After completing this lesson, you will be able to:

- Create a WSDL file with Talend Studio
- Develop SOAP Web service operations
- Develop a SOAP-service-consumer Job
- Access a database with a SOAP Web service

Next step

Make sure you read and understand the lesson slides, then move on to the first lab.

Creating a Simple SOAP Web Service

Overview

This lab introduces you to the process of creating a simple SOAP Web service.

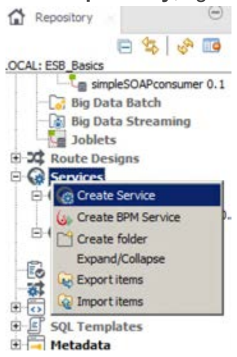
You start by creating a new contract file for the service, called the WSDL file. In this file, you define the service name, its operation name, and the request and response expected elements.

Then you assign a DI Job to your operation.

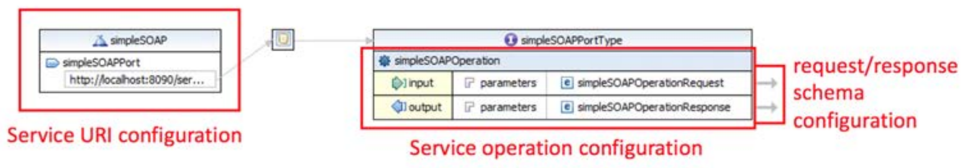
This first Web service is very simple: it takes a string as request parameter and returns this same string in the response message.

Creating the service and its WSDL file

1. In the **Repository**, right-click **Services**. On the contextual menu, choose **Create Service**.

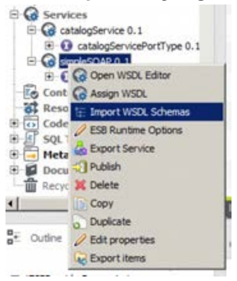


2. In the**Services** wizard window, name your new service **simpleSOAP** and click **Next**.
3. You have a choice: use an existing WSDL file or create a new one. Since you do not have a file, select the **Create new WSDL** option and click **Finish**.
4. The WSDL editor opens, displaying several configuration items:

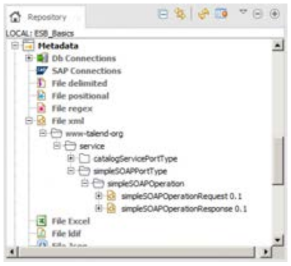

simpleSOAP
simpleSOAPPort
http://localhost:8090/ser...
Service URI configuration
simpleSOAPPortType
simpleSOAPOperation
input
parameters
simpleSOAPOperationRequest
output
parameters
simpleSOAPOperationResponse
Service operation configuration
request/response schema configuration

You can edit any element by double-clicking it. For now, leave the configuration as it is by default.

5. To open the schema editor, next to **simpleSOAPOperationRequest**, click the arrow. This interface allows you to edit, add, and remove elements in the request and response schemas. As you can see, the request schema is very simple. It is expecting a String element in a parameter named "in". Close the schema editor window. Open the response schema and note that the expected response is a unique String element named "out". You will create more-complex schemas later in this chapter. Save and close the WSDL editor.
6. In the **Repository**, right-click your service **simpleSOAP**, and, on the contextual menu, choose **Import WSDL Schemas**.



This operation extracts the request and response schemas from the service operations and stores them as XML metadata in the Metadata section of the repository.
In the **Repository**, navigate to **Metadata > File XML > www-talend-org > service > simpleSOAPPortType > SimpleSOAPOperation**. Inspect the **Request and Response** schema.

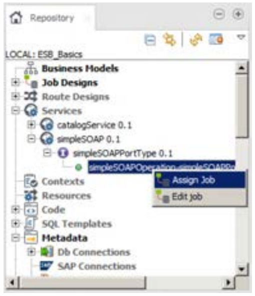


Assigning a DI Job to an operation

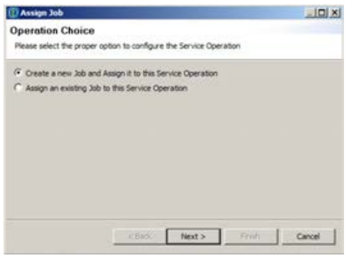
So far, you have created the WSDL contract called the WSDL file, and you have exported your request and response schema as metadata that you can easily use in DI Jobs.

Now you must design the Job that actually does something when the Web service is called.

1. In the **Repository**, navigate to **Services > simpleSOAP 0.1 > simpleSOAPPortType 0.1** and right-click the operation **simpleSOAPOperation-simpleSOAPPortType_simpleSOAPOperation 0.1**. On the menu, select **Assign Job**.



2. In the **Assign Job** wizard window, choose **Create a new Job and Assign it to this Service Operation**, then click **Next**. Note that in the case of a multiple-operation service, you assign a Job to each operation.


Assign Job
Operation Choice
Please select the proper option to configure the Service Operation
Create a new Job and Assign it to this Service Operation
Assign an existing Job to this Service Operation
< Back
Next >
Finish
Cancel

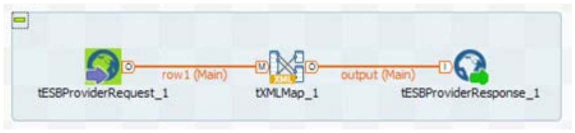
3. In the **New Job** window, keep the default name for your Job and click **Finish**. This creates and opens your new Job, which is now linked to your service operation. It runs when the operation is called via the Web service.
4. New Jobs assigned to a service operation are created with two components by default: tESBProviderRequest and tESBProviderResponse.

   tESBProviderRequest is in charge of receiving the request from the Web service client and transmitting it to the Job.

   tESBProviderResponse is in charge of taking the response message from the Job and sending it back to the Web service client as the service response.

   In the new Job, between the **tESBProviderRequest** and **tESBProviderResponse** components, add a **tXMLMap** component.

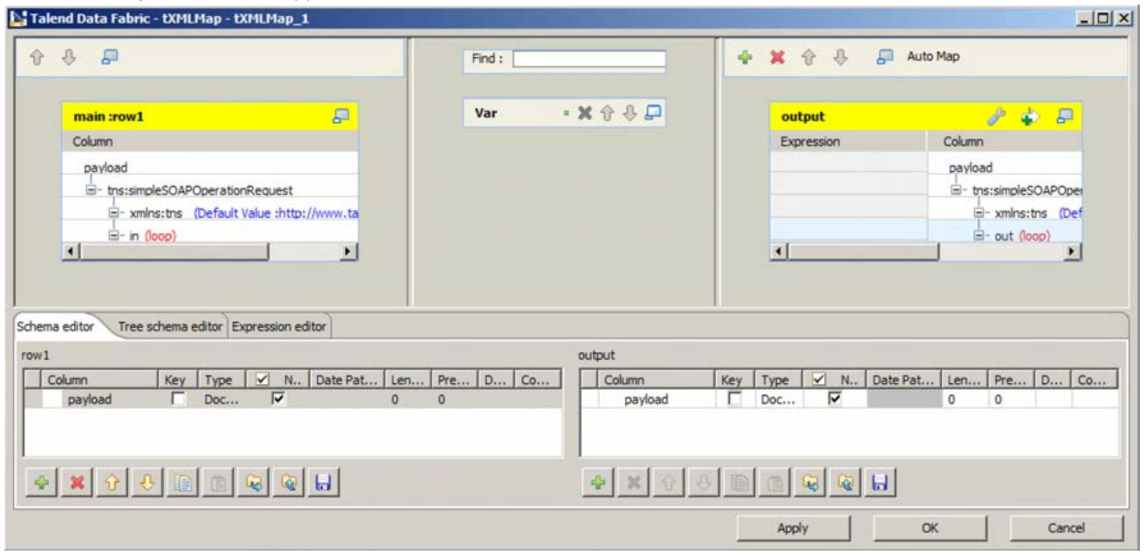
5. Draw a **Main** row from **tESBProviderRequest_1** to **tXMLXMap_1**, and a second one, named *output*, from **tXMLMap_1** to **tESBProviderResponse_1**. When asked if you **want to get the schema of the target component**, choose **Yes**.



6. Double-click the **tXMLMap** component.
7. As you can see, both the **tESBProviderRequest** and **tESBProviderResponse** components share the same schema: a unique Document-type element called *payload*. The payload element holds the SOAP request message (in the tESBProviderRequest component) and the SOAP response message (in the tESBProviderResponse component). SOAP messages are XML messages, so you use:

   a tXMLMap component to extract information from the XML document, process it, and create a new XML document

   the XML metadata of the request and response formats, which provides the schema to extract the request and the schema to build the response
8. In the **tXMLMap window**, right-click the **payload** element in the incoming flow (**row1**). On the menu, choose **Import From Repository**.
9. Navigate to **Metadata > File XML > www-talend-org > service > simpleSOAPPortType > simpleSOAPOperation**, select **simpleSOAPOperationRequest 0.1,** and click **OK**. This imports the request schema of your operation into the payload Document element.
10. Repat the process for the outgoing payload element: right-click the **payload** element in the outgoing flow (**output**), and, on the menu, choose **Import From Repository**.
11. Navigate to **Metadata > File XML > www-talend-org > service > simpleSOAPPortType > simpleSOAPOperation**, select **simpleSOAPOperationResponse 0.1**, and click **OK**.
12. Your **tXMLMap** editor should appear as in the screenshot below:



row1

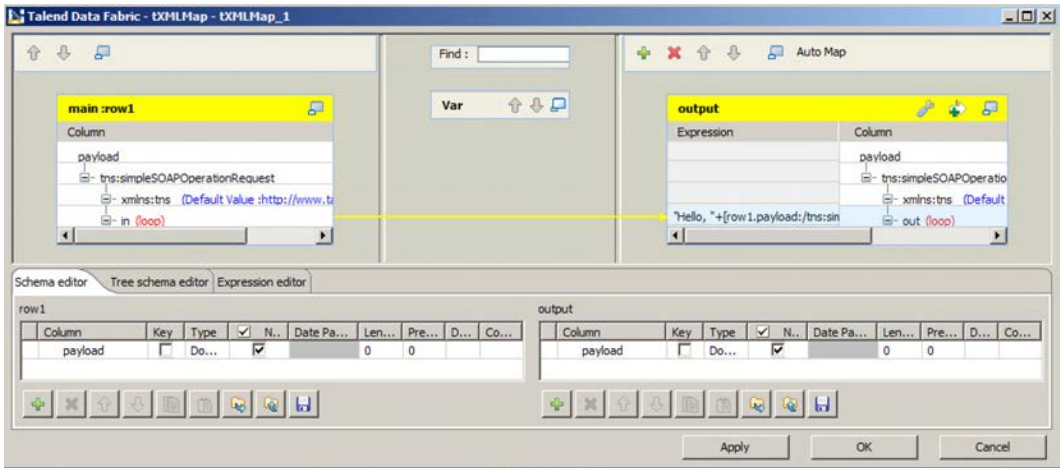
Column	Key	Type	✓ N..	Date Pat...	Len...	Pre...	D...	Co...
payload	☐	Doc...	✓		0	0		

output

Column	Key	Type	✓ N..	Date Pat...	Len...	Pre...	D...	Co...
payload	☐	Doc...	✓		0	0		

13. In the **expression** field of the **out** column in the **output** flow, enter:

   "Hello, "+[row1.payload:/tns:simpleSOAPOperationRequest/in]+"! I hope you're having fun ;)"

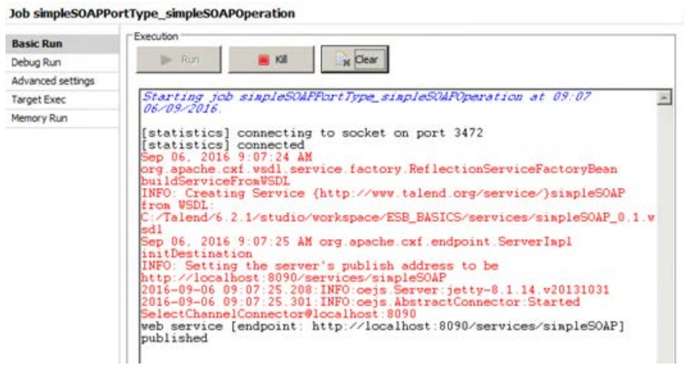


14. To close the tXMLMap editor, click **OK**.

Running the service

Your simple service is ready: it can take an XML document with a unique string as parameter. It answers by repeating the string element in a sentence in the response XML document. You are ready to run and test the Web service.

1. Run the service.



```
Starting job simpleSOAPPortType_simpleSOAPOperation at 09:07
06/09/2016.

[statistics] connecting to socket on port 3472
[statistics] connected
Sep 06, 2016 9:07:24 AM
org.apache.cxf.wsdl.service.factory.ReflectionServiceFactoryBean
buildServiceFromWSDL
INFO: Creating Service {http://www.talend.org/service/}simpleSOAP
from WSDL:
C:/Talend/6.2.1/studio/workspace/ESB_BASICS/services/simpleSOAP_0.1.w
sdl
Sep 06, 2016 9:07:25 AM org.apache.cxf.endpoint.ServerImpl
initDestination
INFO: Setting the server's publish address to be
http://localhost:8090/services/simpleSOAP
2016-09-06 09:07:25.208:INFO:oejs.Server:jetty-8.1.14.v20131031
2016-09-06 09:07:25.301:INFO:oejs.AbstractConnector:Started
SelectChannelConnector@localhost:8090
web service [endpoint: http://localhost:8090/services/simpleSOAP]
published
```

   As you can see from the log message, the service is deployed at *http://localhost:8090/services/simpleSOAP*.

2. To test the service, on your desktop, double-click the **Soap UI 5.2.0** icon. This client tool allows developers to access and test Web services.

3. In **Soap UI**, in the **Navigator** area on the left side of the screen, right-click **Projects**, and, on the menu, select **New SOAP Project**.

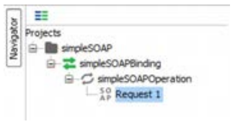
4. In the **New SOAP Project** window, in the **Initial WSDL** text box, enter:

   http://localhost:8090/services/simpleSOAP?wsdl

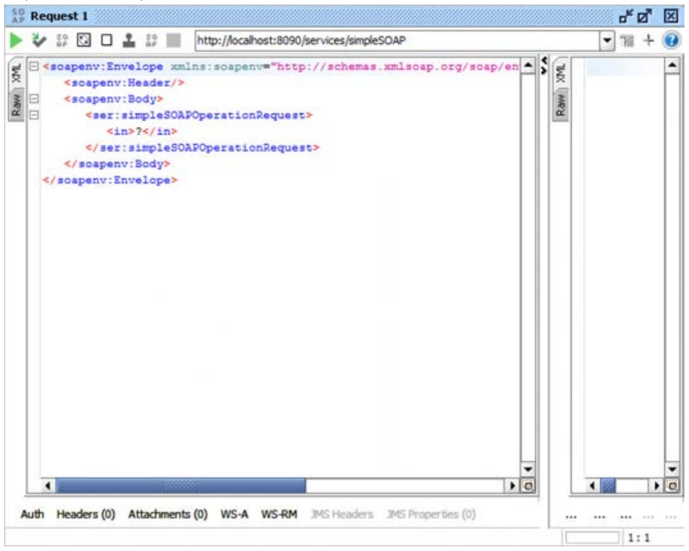
   Click **OK**.

   Providing the WSDL file URL allows the SoapUI client to retrieve all the details of the service. You can find the WSDL file URL by adding *?wsdl* to the service endpoint location.

5. In the new project, navigate to **simpleSOAP > simpleSOAPBinding > simpleSOAPOperation >Request 1**, and double-click **Request 1**.

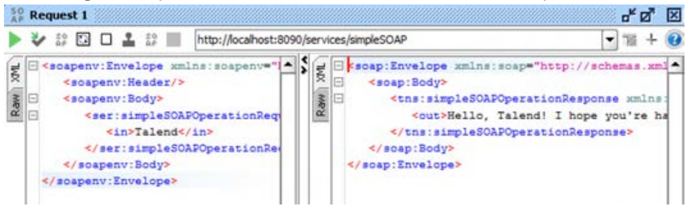


This opens a new **Request** window containing a prebuilt request that you can send to your Web service. Parameters are replaced with question marks.



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/en
   <soapenv:Header/>
   <soapenv:Body>
      <ser:simpleSOAPOperationRequest>
         <in>?</in>
      </ser:simpleSOAPOperationRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

6. In the **Request 1** window, in the parameter tag **in**, replace the question mark with your first name.
7. To send your request to the Web service, click the green **Play** button.
8. On the right side panel, the Web service should send its response.



```
<soapenv:Envelope xmlns:soapenv="
   <soapenv:Header/>
   <soapenv:Body>
      <ser:simpleSOAPOperationReq
         <in>Talend</in>
      </ser:simpleSOAPOperationRe
   </soapenv:Body>
</soapenv:Envelope>
```

```
<soap:Envelope xmlns:soap="http://schemas.xml
   <soap:Body>
      <tns:simpleSOAPOperationResponse xmlns:
         <out>Hello, Talend! I hope you're ha
      </tns:simpleSOAPOperationResponse>
   </soap:Body>
</soap:Envelope>
```

Your service is up and running. Move on to the next lab to create a DI Job to consume this service.

Developing a SOAP Consumer Job

Overview

Talend allows you to develop Web services. It also lets you develop Jobs that consume and use your Web services. This lab shows how you can consume one of your Web services inside another DI Job.

Using the tESBConsumer component

1. Create a new DI Job and name it *simpleSOAPConsumer*.
2. Add the following components to your design:

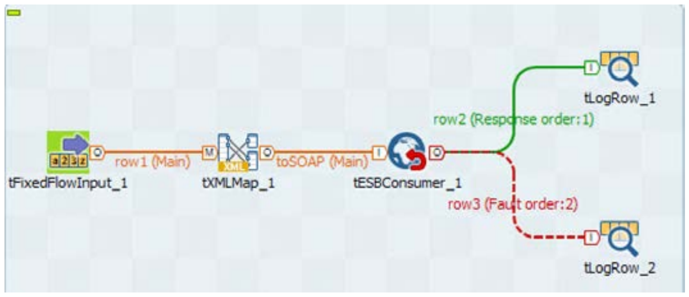
   tFixedFlowInput (generates a fixed input row of data)

   tXMLMap (turns the raw data into an XML SOAP request)

   tESBConsumer (consumes a Web service)

   tLogRow (displays a flow in the console); add two of these

   Link them as shown in the screenshot below:

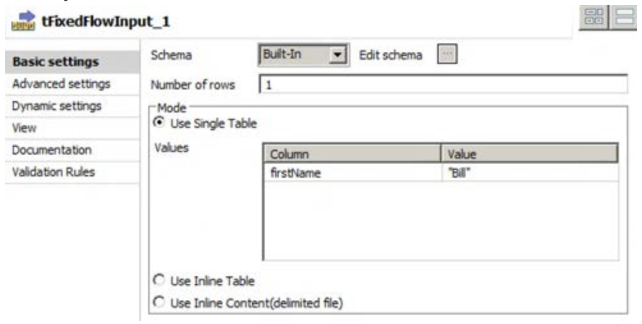


   Notice that two different rows are available as tESBConsumer outputs: **Response**, used to carry the Web service response, and **Fault**, in which you receive any error messages from the Web service.
3. Configure the **tFixedFlowInput** component:

   In the **schema**, create a column named *firstName* of type *String*

   In the **Mode** parameters, select **Use Single Table**.

   To set your **firstName** column, In the **Values** area, in the **Value** column, enter *"Bill"* (or any other name).



Schema: Built-In

Number of rows: 1

Mode

(●) Use Single Table

Values

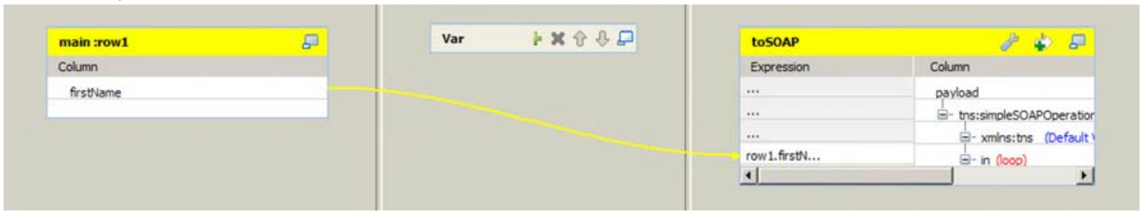
Column	Value
firstName	"Bill"

( ) Use Inline Table

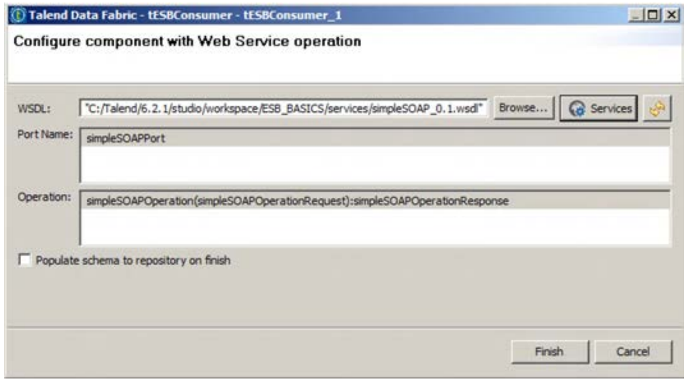
( ) Use Inline Content(delimited file)

4. Double-click the **tXMLMap** component. This is used to turn the single firstName column into an XML message that your Web service can understand. In the **output** flow, right-click the **payload** Document, and, on the menu, choose **Import From Repository**. Navigate to **Metadata > File XML > www-talend-org > service > simpleSOAPPortType > SimpleSOAPOperation**, select **simpleSOAPOperationRequest 0.1**, and click **OK**.

5. Configure the mapping so that the **firstName** column from your input goes into the **in** parameter of your request. To close the tXMLMap editor, click **OK**.



6. Double-click the **tESBConsumer** component. This opens a **tESBConsumer** wizard. You need to provide the WSDL file so that the component retrieves the service contract information. You can enter a path or URL in the **WSDL** text box, or, if the service you want to access and consume was created with Talend Studio (as applicable here), click the **Services** button.
7. In the **Repository Content** window, select the **simpleSOAP** service and click **OK**.
8. There is only one port and one operation for your service, so you do not need to set any other parameters. If your service were to have more than one port or operation, you would need to select the port and operation you want to call. To close the **tESBConsumer** wizard, click **Finish**.



Talend Data Fabric - tESBConsumer - tESBConsumer_1

Configure component with Web Service operation

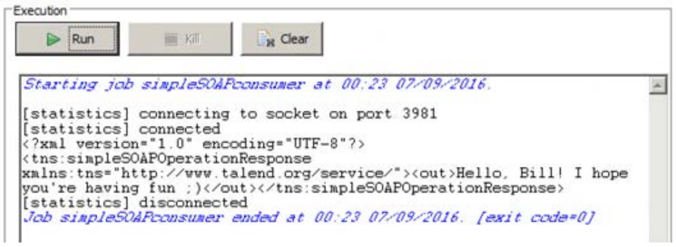
WSDL: "C:/Talend/6.2.1/studio/workspace/ESB_BASICS/services/simpleSOAP_0.1.wsdl"

Port Name: simpleSOAPPort

Operation: simpleSOAPOperation(simpleSOAPOperationRequest):simpleSOAPOperationResponse

☐ Populate schema to repository on finish

9. Run the Job. It should send the firstName you set to the Web service and display the response in the console.



```
Starting job simpleSOAPconsumer at 00:23 07/09/2016.

[statistics] connecting to socket on port 3981
[statistics] connected
<?xml version="1.0" encoding="UTF-8"?>
<tns:simpleSOAPOperationResponse
xmlns:tns="http://www.talend.org/service/"><out>Hello, Bill! I hope
you're having fun ;)</out></tns:simpleSOAPOperationResponse>
[statistics] disconnected
Job simpleSOAPconsumer ended at 00:23 07/09/2016. [exit code=0]
```

Now you know how to create and consume Web services with Talend. In the next lab, you will create a new SOAP service to access a catalog database.

Creating a Product Catalog SOAP Service

Overview

This lab shows you how to build a more complex SOAP Web service with Talend Studio.

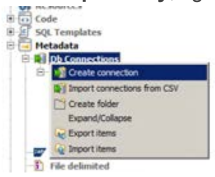
This SOAP Web service is in charge of retrieving information about products in a database. It takes a product ID as parameter and returns a detailed XML message with product information from the database.

As noted at the beginning of this course, this lab requires solid knowledge of DI development.

Configuring database access

First you will create database metadata to easily handle connection to the MySQL schema.

1. In the **Repository**, right-click **Metadata > Db Connections,** and, on the menu, choose **Create connection**.



2. In the wizard, in the **Name** text box, enter *trainingDatabase* and click **Next**.
3. Set the connection parameters as follows:

   DB Type: *MySQL*

   Db Version: *MySQL 5*

   Login: *talend*

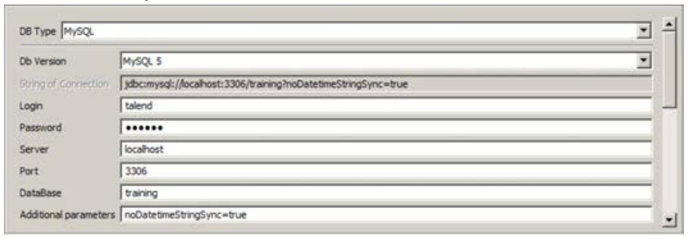
   Password: *talend*

   Server: *localhost*

   Port: *3306*

   DataBase: *training*

   To confirm that you can connect to the database, click **Check**, then **Finish**.



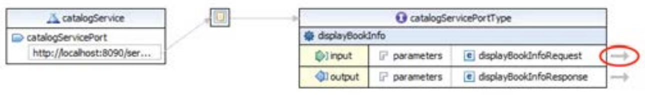
DB Type	MySQL
Db Version	MySQL 5
String of Connection	jdbc:mysql://localhost:3306/training?noDatetimeStringSync=true
Login	talend
Password	••••••
Server	localhost
Port	3306
DataBase	training
Additional parameters	noDatetimeStringSync=true

4. In the **Repository**, right-click the **trainingDatabase** metadata, and, on the menu, choose **Retrieve Schema**. Retrieve the schema of both the **books** and **movies** tables.
5. Use the **Data Viewer** to inspect the tables. In this lab, you will query the **books** table.

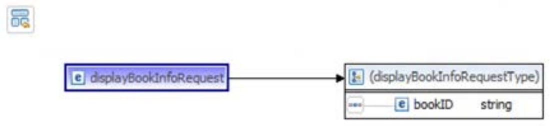
Creating the SOAP service

1. In the **Repository**, right-click **Services**, create a new SOAP service, and name it *catalogService*. Choose to create a new WSDL file for this service.

2. In the WSDL editor, to edit the request format, click the arrow to the right of the input parameters.



3. In the new window, right-click the default **in** parameter and rename it *bookID*.



4. Save, close the window, and go back to the WSDL editor. To edit the reponse format, right-click the arrow to the right of the output parameters.
   The response sends all the columns of information about the book that the database can return. To add new columns to this schema, right-click in the schema and choose **Insert Element**.
   Edit the model so that it holds the following elements:

   title, string

   author, string

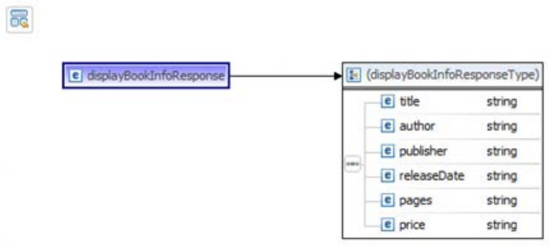
   publisher, string

   releaseDate, string

   pages, string

   price, string

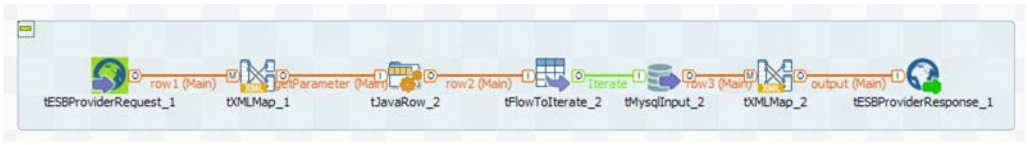
   When you are finished, save and close your schema.



5. In the WSDL editor, save the service.
6. To save your request and response schemas as XML metadata, in the **Repository**, right-click the **catalogService** element and choose **Import WSDL Schemas**.

Creating the Job assigned to the service operation

1. In the **Repository**, navigate to the **catalogService** operation, right-click it, and choose **Assign Job**.
2. In the wizard, create a new Job and keep its default name.
3. The Job behind this service will look like in the screenshot below:



   View the components:

tESBProviderRequest retrieves the request and sends it in the Main flow.

tXMLMap reads the XML request and extracts the bookID parameter from it.

tJavaRow stores the bookID value in a context variable.

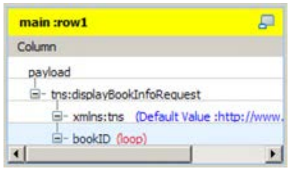
tFlowToIterate breaks the Main flow and allows you to connect to a tMySQLInput.

tMySQLInput runs a query on the database: it retrieves all information columns about the book with the ID that matches the bookID context variable.

tXMLMap takes the raw data from the database and formats it in the XML response message.

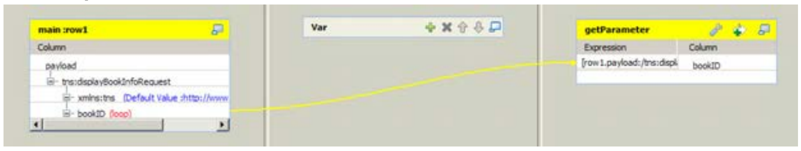
tESBProviderResponse sends the XML response to the Web service client.

4. Add a **tXMLMap** component and link it to the **tESBProviderRequest** component via **Main** row.
5. Double-click the **tXMLMap** component. In the **tXMLMap** window, right-click the **payload** element, and choose **Import From Repository**. Based on what you did in the first lab in this chapter, import the request schema of your service metadata. Your tXMLMap interface should appear as in the screenshot below.



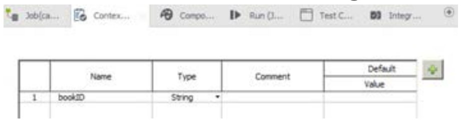
```
main :row1
Column
payload
  tns:displayBookInfoRequest
    xmlns:tns  (Default Value :http://www.
    bookID (loop)
```

6. Still in the **tXMLMap** editor, create an output flow and name it *getParameter*.
7. Add a single column to this output flow, name it *bookID*, and assign it the **String** type.
8. Map the **bookID** from the input into the **bookID** column from the **getParameter** output, then click **OK** to save and close the **tXMLMap** editor.


main :row1
Column
payload
tns:displayBookInfoRequest
xmlns:tns (Default Value :http://www
bookID (loop)
Var
getParameter

Expression	Column
[row1.payload:/tns:displ	bookID

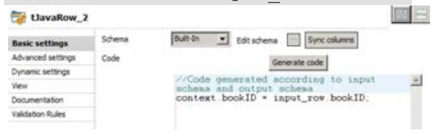
9. Add a **tJavaRow** component to the right of the **tXMLMap** component. Right-click the **tXMLMap** component, select the **Row > getParameter** output, and link it to the **tJavaRow** component.
10. In the **Context** view, add a new **String** context variable and name it *bookID*.



	Name	Type	Comment	Default Value
1	bookID	String		

11. In the **tJavaRow Component** view, write this piece of code to send the result of the tXMLMap into the context variable:

```
context.bookID = input_row.bookID;
```



Schema: Built-In

Edit schema

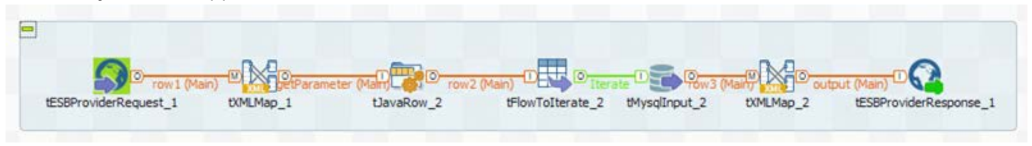
Code:

```
//Code generated according to input schema and output schema
context.bookID = input_row.bookID;
```

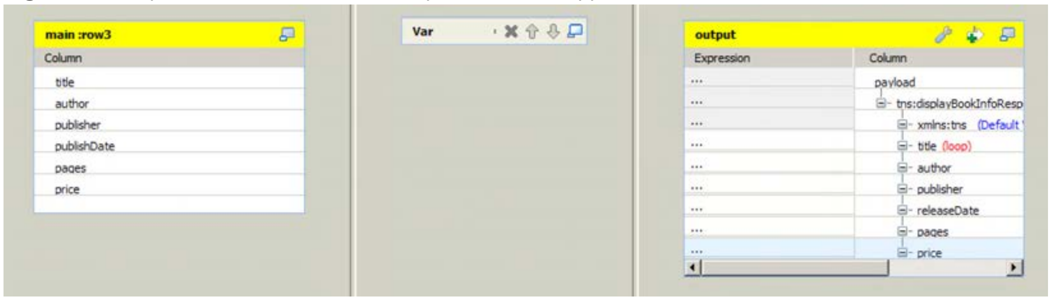
12. Add a **tFlowToIterate** component and draw a **Main** row to it from the **tJavaRow** component.
13. In the **Repository**, navigate to **Metadata > Db Connections > trainingDatabase 0.1 > Table Schemas**. Drag and drop the **books** schema to the right of the **tFlowToIterate** component. When asked which component to use, choose **tMySQLInput**.
14. From the **tFlowToIterate** component, draw an **Iterate** link to the **tMySQLInput** component.
15. In the **tMySQLInput Component** view, you must update the SQL query so that it only selects data for the requested bookID. You do this by adding a WHERE condition. Update the SQL query as follows:

```
"SELECT `books`.`title`, `books`.`author`, `books`.`publisher`, `books`.`publishDate`, `book-
s`.`pages`, `books`.`price` FROM `books` where `books`.`bookID` = '"+context.bookID+"'"
```

16. Add a final **tXMLMap** component. Link it to the **tMySQLInput** component in input, and the **tESBProviderResponse** component in output. Name the output *output,* and, when asked if you want to get the schema of the target component, click **Yes**. Your job should appear as in the screenshot below.



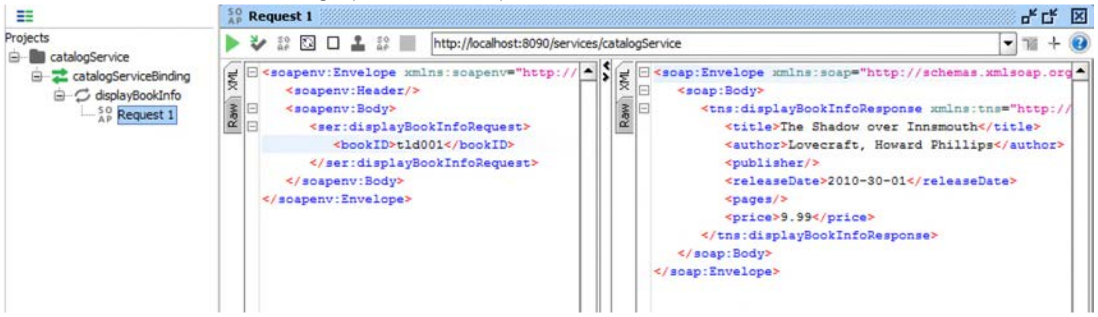
17. Open the **tXMLMap_2** component. In the **tXMLMap** editor, your input flow should show the columns returned by the SQL query, whereas the output flow should have one payload element.
18. Right-click the **payload** element, choose **Import From Repository**, and import the XML metadata of your **catalogService** response schema. Your tXMLMap editor should appear as follows:



19. Map each column from the **input** flow into its column in the **output** schema (or to save time, choose **Auto Map**). To save and close your **tXMLMap**, click **OK**.

Running and testing the service

1. Run the job.
2. In **Soap UI**, create a new SOAP project.
3. Provide the SOAP project with the WSDL URL, *http://localhost:8090/services/catalogService?wsdl*
4. In the **Request** window, set the **bookID** parameter to *tld001* and click the **Play** button to send your request to the service.
5. The service should answer in the right panel of the Request window.



```
<soapenv:Envelope xmlns:soapenv="http://
   <soapenv:Header/>
   <soapenv:Body>
      <ser:displayBookInfoRequest>
         <bookID>tld001</bookID>
      </ser:displayBookInfoRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org
   <soap:Body>
      <tns:displayBookInfoResponse xmlns:tns="http://
         <title>The Shadow over Innsmouth</title>
         <author>Lovecraft, Howard Phillips</author>
         <publisher/>
         <releaseDate>2010-30-01</releaseDate>
         <pages/>
         <price>9.99</price>
      </tns:displayBookInfoResponse>
   </soap:Body>
</soap:Envelope>
```

Next step

You have almost completed this lesson. Continue to the Wrap-up section for a review of the concepts we covered.

Wrap-up

Now you know more about SOAP Web services and how to design them in Talend Studio.

You can create your own WSDL file, including operations and request/reponse formats. You know how to assign each operation a DI Job, and you can develop DI Jobs with the components tESBProviderRequest and tESBProviderResponse.

In Web-service operation-assigned Jobs, you know how to use XML schemas and the tXMLMap component to parse a request and create a response message. You can also design a rich service that takes advantage of several DI components.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let’s continue >**.

LESSON 10

Developing REST Web Services

This chapter discusses the following.

Overview

This section guides you in designing Jobs that deploy as REST Web services. While SOAP is operation oriented, REST is resource oriented and leverages HTTP methods ("verbs") GET, PUT, POST, and DELETE. Unlike SOAP, REST does not require a contract file.

The following labs create REST Web services that access a resource with different URIs and different HTTP verbs.

Objectives

After completing this lesson, you will be able to:

- Design a REST Web service and use the dedicated REST components
- Add more than one endpoint to a REST service
- Map several HTTP verbs to the same URI endpoint
- Consume a REST service

Next step

Make sure you read and understand the lesson slides, then move on to the first lab.

Creating a Simple REST Web Service

Overview

This lab shows you how to create a simple REST Web service.

Unlike the SOAP standard, a REST Web service does not need a contract, as the REST standard defines all the operations you can invoke on a resource.

As you learned from the slides, a resource is exposed on a URI. To request a REST service, you need to provide this URI along with an HTTP verb. Each HTTP verb represents a possible operation that you can run on the resource. The four HTTP verbs are:

GET—runs a READ operation on the resource

POST—runs a CREATE operation on the resource

PUT—runs an UPDATE operation on the resource

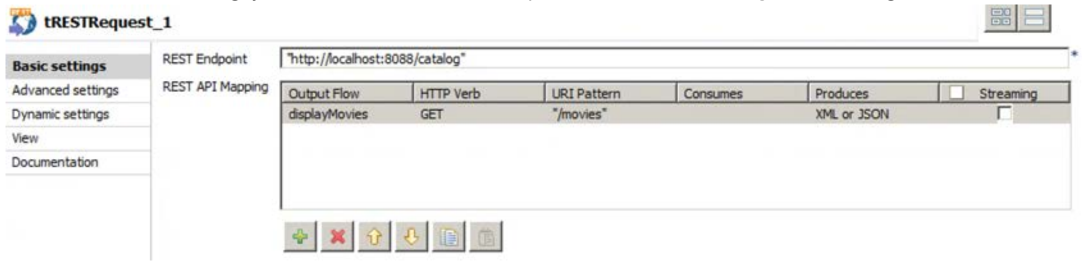
DELETE—runs a DELETE operation on the resource

In this first lab, you will develop a catalog REST service that is very similar to the one you developed with the SOAP standard.

Developing the REST service

1. In Talend Studio, in the **Repository**, create a new **Standard Job** and name it *RESTCatalog_v1*.
2. Add a **tRESTRequest** component. On the developer side, this component defines your REST service endpoint, URI patterns, and HTTP verbs. On the deployment side, it is also responsible for exposing the service in your Talend environment.
3. In the **tRESTRequest Component** view, in the **REST endpoint** text box, enter *"http://localhost:8088/catalog"*. This parameter sets the base URL where your REST service is deployed and accessible.
4. To create the part of your service that allows a client to access the movie catalog, click the **Plus (+)** button and add a new line in the **REST API mapping** section. Each new line adds an operation to your REST service.
   a. On the new line of the **REST API Mapping** section, in the **URI pattern** column, enter *"/movies"*
   b. In the **HTTP Verb** column, make sure **GET** is selected.
   c. In the **Output Flow** column, click the **Ellipsis (...)** button, enter *displayMovies* and click **OK**. A new window opens, permitting you to define a schema for the output. Leave it empty and click **OK**.

   This configuration means that you created a new REST service, and it deploys on the URL *http://localhost:8088/catalog*
   The catalog service offers a movies resource with the HTTP verb GET. This resource can be accessed by concatenating the service URL and resource URI pattern. The URI pattern is **"/movies"**
   To read the movie catalog, you invoke the service URL *http://localhost:8088/catalog/movies* along with the verb GET.



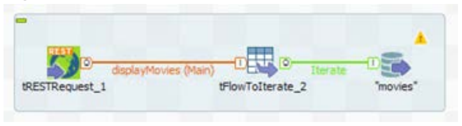
REST Endpoint: "http://localhost:8088/catalog"

REST API Mapping:

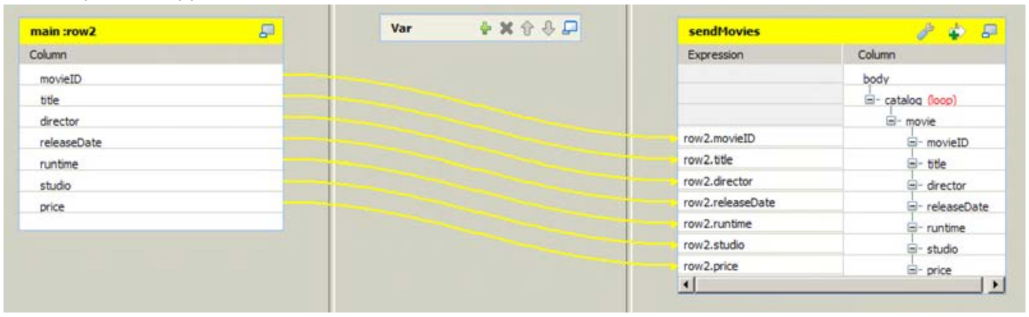
Output Flow	HTTP Verb	URI Pattern	Consumes	Produces	Streaming
displayMovies	GET	"/movies"		XML or JSON	☐

5. When a client queries the movies resource in your catalog service with the verb GET, you are expected to read the movie table entries in your MySQL database and return them in an XML answer message. The Job you need to create is simple: read the movie table in your database, format the answer as XML, and send it back to the client.
6. You need to read data from a MySQL database, which means you need to use a tMySQLInput component. This component does not accept a Main row connection as an input. Unfortunately, the tRESTRequest component offers only a Main row as a connector. As a result, you use a tFlowToIterate component, which can break a Main row and offer an iterate trigger.
   Add a **tFlowToIterate** component next to the **tRESTRequest** component.

7. Right-click the **tRESTRequest** component. On the contextual menu, find the **displayMovies** flow that you defined in the parameters in the **Row** list. Select this output and link it to the **tFlowToIterate** component.
8. From the **Repository**, navigate to **Metadata > Db Connections > trainingDatabase 0.1 > Table schemas > movies**. Drag and drop the **movies** metadata in the design area and choose to use it with a **tMySQLInput** component.
9. Right-click the **tFlowToIterate** component, select the **Iterate** row from the menu, and link it to the **"movies"** component.



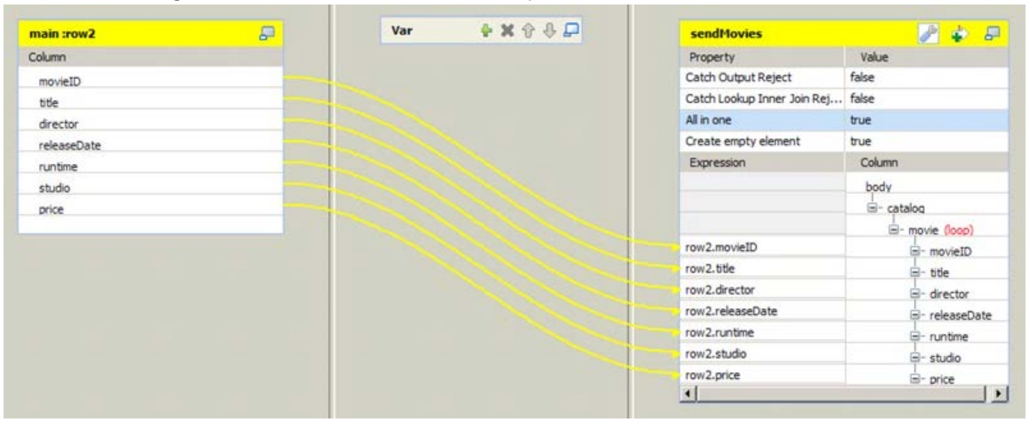
10. To create the XML response message, add a **tXMLMap** component to the right of the **"movies" tMySQLInput** component, then link those two components via a Main row.
11. The response message is sent to the client by a component named **tRESTResponse**. Place a **tRESTResponse** component after the **tXMLMap** component.
12. Link **tXMLMap** to the **tRESTResponse** component and name the output *sendMovies*. When asked if you want the schema of the target component, choose **Yes**.
13. Double-click **tXMLMap**.
14. In the **tXMLMap** editor, in the **sendMovies** output, you can see a unique Document element named **body**. This is the default, empty response of the **tRESTResponse** component. Under **body**, right-click the **root** element and choose **Rename** from the menu. Rename the root element *catalog*.
15. The **catalog** element is now the root tag of the XML document. A catalog contains several movie items. Right-click **catalog** and select **Create Sub-Element**. As the new label, enter **movie** and click **OK**.
16. To create the structure of the movie item, select all elements from the **input (main:row2)**, then drag and drop them onto the **movie** element. When asked how to create these elements, select **Create as sub-element of target node**. Your tXMLMap should appear as in the screenshot below.



Column
movieID
title
director
releaseDate
runtime
studio
price

Expression	Column
	body
	catalog (loop)
	movie
row2.movieID	movieID
row2.title	title
row2.director	director
row2.releaseDate	releaseDate
row2.runtime	runtime
row2.studio	studio
row2.price	price

17. The loop element of your XML structure is still on the catalog tag, whereas the loop is now on the movie tag. To change the looping element, right-click **movie**, choose **As loop element** from the menu, and click **OK**.
18. Open the **tXMLMap** settings (the tool button above the **sendMovies** box). To aggregate all lines of data coming from the database into a single XML document, set the **All in one** parameter to *true*.



Column
movieID
title
director
releaseDate
runtime
studio
price

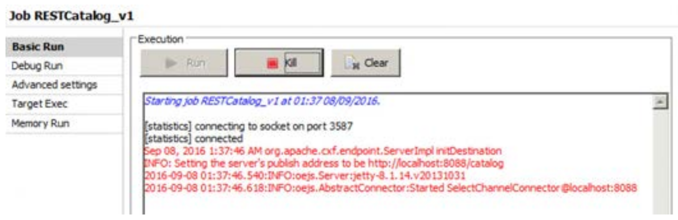
Property	Value
Catch Output Reject	false
Catch Lookup Inner Join Rej...	false
All in one	true
Create empty element	true

Expression	Column
	body
	catalog
	movie (loop)
row2.movieID	movieID
row2.title	title
row2.director	director
row2.releaseDate	releaseDate
row2.runtime	runtime
row2.studio	studio
row2.price	price

19. To close the tXMLMap editor, click **OK**. Your REST service is now ready to run.

Running the service

1. Run the service.

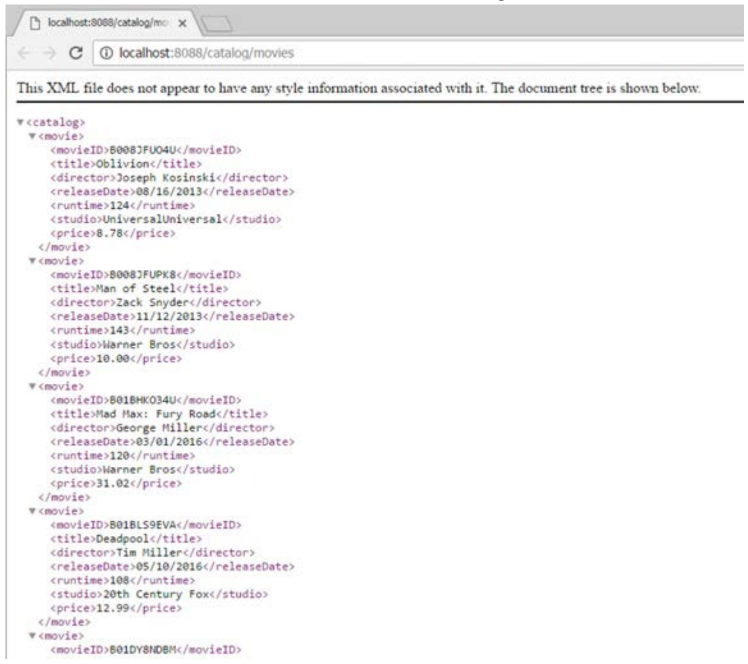


```
Starting job RESTCatalog_v1 at 01:37 08/09/2016.

[statistics] connecting to socket on port 3587
[statistics] connected
Sep 08, 2016 1:37:46 AM org.apache.cxf.endpoint.ServerImpl initDestination
INFO: Setting the server's publish address to be http://localhost:8088/catalog
2016-09-08 01:37:46.540:INFO:oejs.Server:jetty-8.1.14.v20131031
2016-09-08 01:37:46.618:INFO:oejs.AbstractConnector:Started SelectChannelConnector@localhost:8088
```

   As you can see from the INFO log message, the service is published at *http://localhost:8088/catalog.*

2. To test the service, open a Web browser.
3. In the navigation bar, enter *http://localhost:8088/catalog/movies*. You do not need to specify the GET verb (Web browsers send it by default).
4. The REST service should answer with an XML message similar to the one in the screenshot below:



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<catalog>
  ▼<movie>
     <movieID>B008JFUO4U</movieID>
     <title>Oblivion</title>
     <director>Joseph Kosinski</director>
     <releaseDate>08/16/2013</releaseDate>
     <runtime>124</runtime>
     <studio>UniversalUniversal</studio>
     <price>8.78</price>
   </movie>
  ▼<movie>
     <movieID>B008JFUPK8</movieID>
     <title>Man of Steel</title>
     <director>Zack Snyder</director>
     <releaseDate>11/12/2013</releaseDate>
     <runtime>143</runtime>
     <studio>Warner Bros</studio>
     <price>10.00</price>
   </movie>
  ▼<movie>
     <movieID>B01BHKO34U</movieID>
     <title>Mad Max: Fury Road</title>
     <director>George Miller</director>
     <releaseDate>03/01/2016</releaseDate>
     <runtime>120</runtime>
     <studio>Warner Bros</studio>
     <price>31.02</price>
   </movie>
  ▼<movie>
     <movieID>B01BLS9EVA</movieID>
     <title>Deadpool</title>
     <director>Tim Miller</director>
     <releaseDate>05/10/2016</releaseDate>
     <runtime>108</runtime>
     <studio>20th Century Fox</studio>
     <price>12.99</price>
   </movie>
  ▼<movie>
     <movieID>B01DY8NDBM</movieID>
```

Adding a second URI pattern

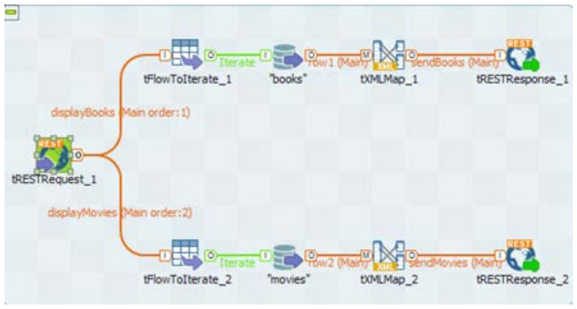
1. In the **Component** view of the **tRESTRequest** component, click the **Plus (+)** button to add a new line in the **REST API Mapping** section, and set the following parameters:

   URI pattern: *"/books"*

   HTTP Verb: *GET*

   Output Flow: *displayBooks*

2. Repeating steps 6 – 19 from "Developing the REST service," create a second path for your REST service so that it can also return the books information from the database.



When your service is up and running, move on to the next lab to create a DI Job that consumes REST resources.

Developing a REST Consumer Job

Overview

This lab shows how you can consume one of your REST Web services inside another DI Job.

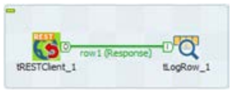
Using the tRESTClient component

1. Create a new DI Job and name it *RESTConsumer_v1*.
2. Add the following components to your design:

   tRESTClient (consumes a REST service)

   tLogRow (displays the flow in the console)

   Link the **tRESTClient** component to the **tLogRow** component using the **Response** row:

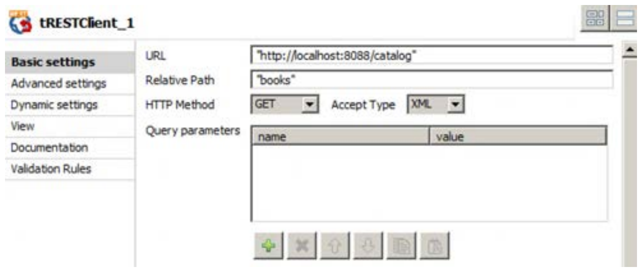
   

   Two different rows are available as tRESTClient outputs: **Response**, used to carry the Web service response, and **Error**, in which you receive the error messages.
3. Configure the **tRESTClient** component:

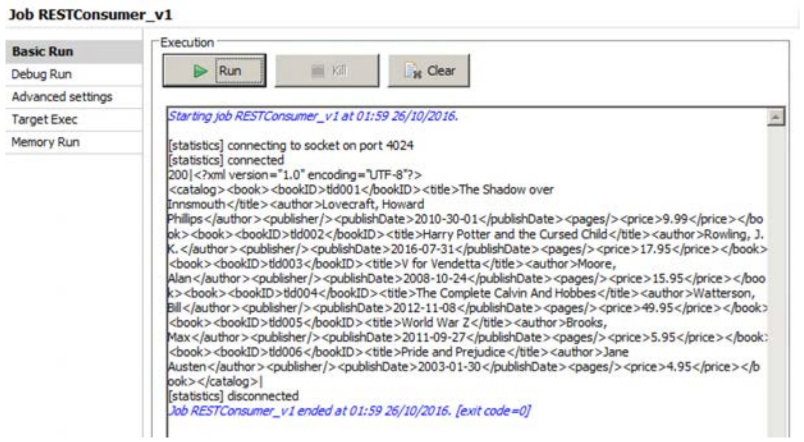
   In the **URL** text box, enter *"http://localhost:8088/catalog"*

   In the **Relative Path** text box, enter *"books"*

   In the **HTTP Method** menu, select **GET**

   URL: "http://localhost:8088/catalog"
   Relative Path: "books"
   HTTP Method: GET
   Accept Type: XML
4. Run the Job. The REST service response should show in the console.

```
Starting job RESTConsumer_v1 at 01:59 26/10/2016.

[statistics] connecting to socket on port 4024
[statistics] connected
200|<?xml version="1.0" encoding="UTF-8"?>
<catalog><book><bookID>tld001</bookID><title>The Shadow over
Innsmouth</title><author>Lovecraft, Howard
Phillips</author><publisher/><publishDate>2010-30-01</publishDate><pages/><price>9.99</price></bo
ok><book><bookID>tld002</bookID><title>Harry Potter and the Cursed Child</title><author>Rowling, J.
K.</author><publisher/><publishDate>2016-07-31</publishDate><pages/><price>17.95</price></book>
<book><bookID>tld003</bookID><title>V for Vendetta</title><author>Moore,
Alan</author><publisher/><publishDate>2008-10-24</publishDate><pages/><price>15.95</price></boo
k><book><bookID>tld004</bookID><title>The Complete Calvin And Hobbes</title><author>Watterson,
Bill</author><publisher/><publishDate>2012-11-08</publishDate><pages/><price>49.95</price></book>
<book><bookID>tld005</bookID><title>World War Z</title><author>Brooks,
Max</author><publisher/><publishDate>2011-09-27</publishDate><pages/><price>5.95</price></book:
<book><bookID>tld006</bookID><title>Pride and Prejudice</title><author>Jane
Austen</author><publisher/><publishDate>2003-01-30</publishDate><pages/><price>4.95</price></b
ook></catalog>|
[statistics] disconnected
Job RESTConsumer_v1 ended at 01:59 26/10/2016. [exit code=0]
```

Now you know how to create and consume REST services with Talend Studio. In the next lab, you will explore more features of the tRESTRequest component.

Adding URL Parameters in a REST Service

Overview

REST services can receive and read parameters in request URLs. This lab shows you how to configure and use these parameters.

You will add a new URI pattern to the REST service you developed. The URI pattern contains a movie (or book) ID, and the service must return information for only this specific item.

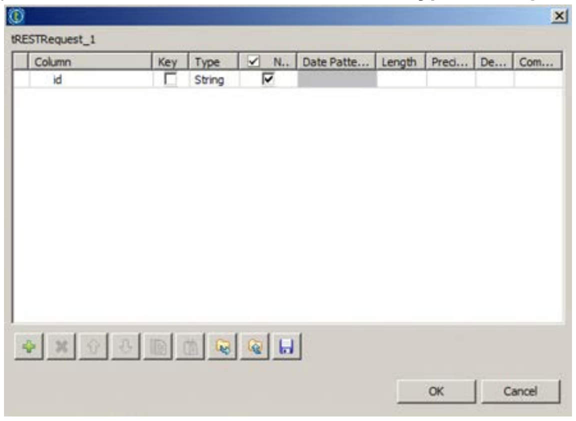
Developing the REST service

1. In the **Repository**, duplicate **RESTCatalog_v1** and name it *RESTCatalog_v2*.
2. Open **RESTCatalog_v2**.
3. In the **tRESTRequest Component** view, create a new **REST API Mapping** line and set the following parameters:

   URI pattern: *"/movie/{id}"*

   HTTP Verb: *GET*

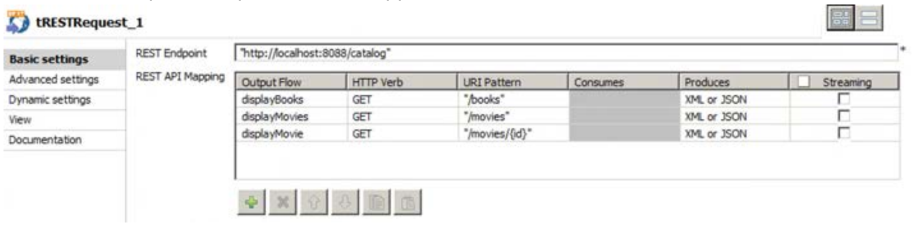
   Notice the **{id}** element in the URI pattern. The brackets tell the tRESTRequest component that there will be a variable element named "id".
4. In the **Output Flow** parameter of your new **REST API Mapping** line, enter *displayMovie*. This time, when you name the **output** flow, when the **Schema editor** window opens, click the **Plus (+)** button to add a column. This flow contains the **id** parameter. Name the column *id* and set the **Type** to *String*.



tRESTRequest_1

Column	Key	Type	N..	Date Patte...	Length	Preci...	De...	Com...
id	☐	String	☑					

5. Your tRESTRequest component should appear as in the screenshot below:



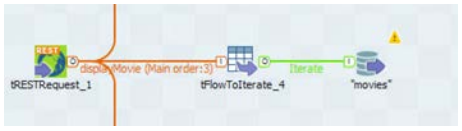
tRESTRequest_1

REST Endpoint: "http://localhost:8088/catalog"

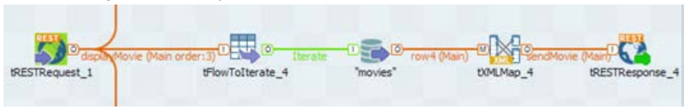
REST API Mapping:

Output Flow	HTTP Verb	URI Pattern	Consumes	Produces	Streaming
displayBooks	GET	"/books"		XML or JSON	☐
displayMovies	GET	"/movies"		XML or JSON	☐
displayMovie	GET	"/movies/{id}"		XML or JSON	☐

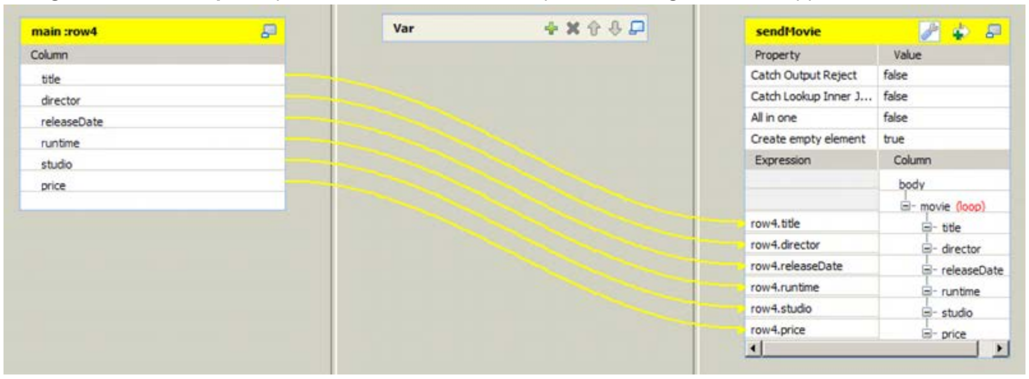
6. In the design area, add a **tFlowToIterate** component and link it to your new **displayMovie** output from the **tRESTRequest** component.
7. In the **Repository**, place the **movies** table metadata in the design area and use it as a new **tMySQLInput** component.
8. Draw an iterate link from the **tFlowToIterate** component to the **"movies"** component.



9. To complete this service operation, add a **tXMLMap** component and a **tRESTResponse** component. Link the **"movies" tMySQLInput** component as your **tXMLMap** input. Create a new output from the **tXMLMap** that connects to the **tRESTResponse** component and name it *sendMovie*



10. Click the **"movies" tMySQLInput** component and go to its **Component** view.
11. The SQL query needs to be updated so that it only returns data for the item ID sent as service parameter. The ID element is propagated from the **tRESTRequest** component via the **tFlowToIterate** component. At the end of the SQL query, remove the double quotes and add:
`WHERE `movies`.`movieID` ='"+((String)globalMap.get("displayMovie.id"))+"'"`
Note that you can easily find the ID value by invoking code completion (**CTRL+SPACEBAR**) in the **tFlowToIterate** variables.
12. Configure the **tXMLMap** component to create the XML response message. It should appear as in the screenshot below:

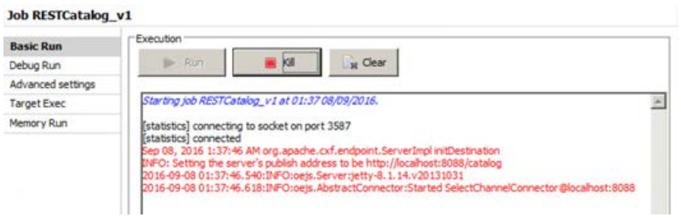


Property	Value
Catch Output Reject	false
Catch Lookup Inner J...	false
All in one	false
Create empty element	true

Expression	Column
	body
	movie (loop)
row4.title	title
row4.director	director
row4.releaseDate	releaseDate
row4.runtime	runtime
row4.studio	studio
row4.price	price

Running the service

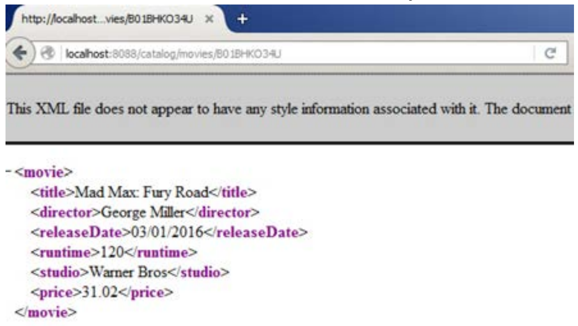
1. Run the service.



```
Starting job RESTCatalog_v1 at 01:37 08/09/2016.

[statistics] connecting to socket on port 3587
[statistics] connected
Sep 08, 2016 1:37:46 AM org.apache.cxf.endpoint.ServerImpl initDestination
INFO: Setting the server's publish address to be http://localhost:8088/catalog
2016-09-08 01:37:46.540:INFO:oejs.Server:jetty-8.1.14.v20131031
2016-09-08 01:37:46.618:INFO:oejs.AbstractConnector:Started SelectChannelConnector@localhost:8088
```

2. In your Web browser, enter *http://localhost:8088/catalog/movies/B01BHKO34U* (or use any movie ID from the catalog).
3. The REST service should return data for only one movie.

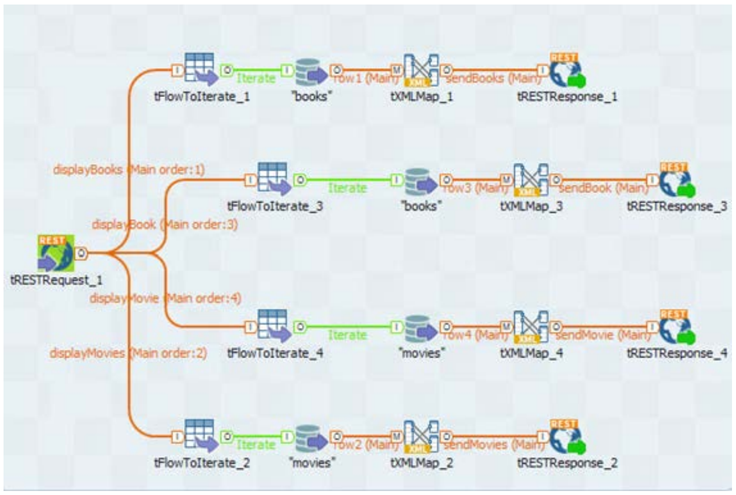

http://localhost...vies/B01BHKO34U
localhost:8088/catalog/movies/B01BHKO34U

This XML file does not appear to have any style information associated with it. The document

```
- <movie>
    <title>Mad Max: Fury Road</title>
    <director>George Miller</director>
    <releaseDate>03/01/2016</releaseDate>
    <runtime>120</runtime>
    <studio>Warner Bros</studio>
    <price>31.02</price>
  </movie>
```

Challenge

Add a fourth operation to your REST service so that you can request catalog information about one book by passing its ID into the request URL.



It is time to learn more about the other HTTP verbs. You can move on to the next lab.

Using the HTTP Verb POST

Overview

So far, you have been using the HTTP verb GET, which lets you read from a resource.

In this lab, you will use the HTTP verb POST. This verb allows to write in the resource so that a POST request always comes with some data attached: the data you want to write to the resource.

You will create a new REST API Mapping line on the /movies URI pattern for the HTTP verb POST. This operation allows a client to write a new movie entry to the database.

Adding a new operation to the REST service

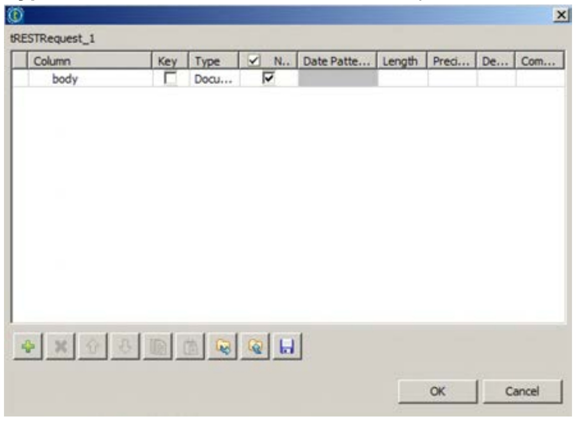
1. In the **Repository**, duplicate **RESTCatalog_v2** and name it *RESTCatalog_v3*.
2. Open **RESTCatalog_v3**.
3. In the **tRESTRequest Component** view, create a new **REST API Mapping** line and set the following parameters:

   URI pattern: *"/movies"*

   HTTP Verb: *POST*

   Output Flow: *addMovie*

4. When the Schema editor window opens, click the **Plus (+)** button to add a new column. Name the column *body* and set the **Type** to *Document*. This indicates to the component that the POST information is contained in the body of a message.

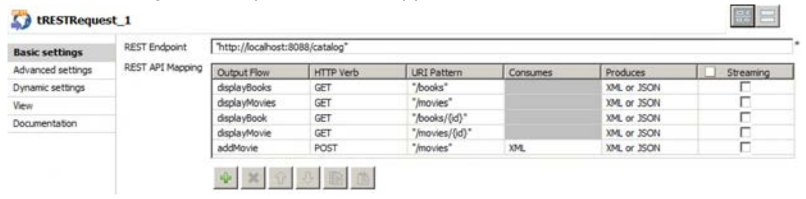
   

   tRESTRequest_1

Column	Key	Type	N..	Date Patte...	Length	Preci...	De...	Com...
body	☐	Docu...	☑					

   To close the Schema editor, click **OK**.
5. The document type you set in the schema can be JSON or XML. However, in the **Consumes** column, you can choose a specific type of document. Set the **Consumes** parameter to *XML* only.

6. Your **tRESTRequest** component should appear as in the screenshot below:

   tRESTRequest_1

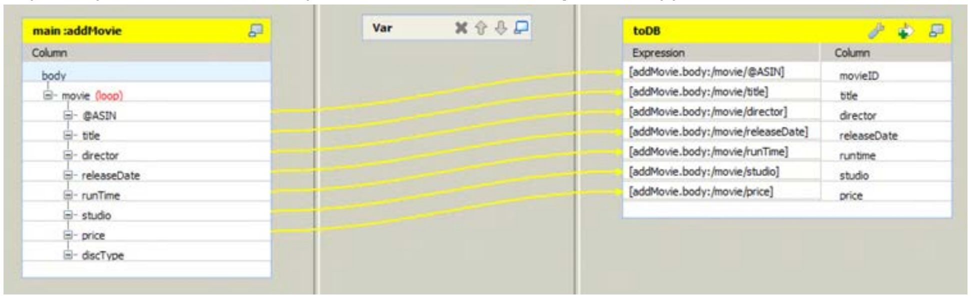
   REST Endpoint: "http://localhost:8088/catalog"

   REST API Mapping

Output Flow	HTTP Verb	URI Pattern	Consumes	Produces	Streaming
displayBooks	GET	"/books"		XML or JSON	☐
displayMovies	GET	"/movies"		XML or JSON	☐
displayBook	GET	"/books/{id}"		XML or JSON	☐
displayMovie	GET	"/movies/{id}"		XML or JSON	☐
addMovie	POST	"/movies"	XML	XML or JSON	☐

7. The schema of the XML document that you will send to the service is the same as the one you used in the Camel route labs. For reference, open the file *C:\StudentFiles\routes\r09_multicast\movie6.xml*.
8. From the **tRESRequest** component, you get a Main row containing an XML document. You just need to insert the data into the movies database and then return a message to the service client. Add a **tXMLMap** component to the design area and link the **addMovie** output from the **tRestRequest** component to the **tXMLMap** component.

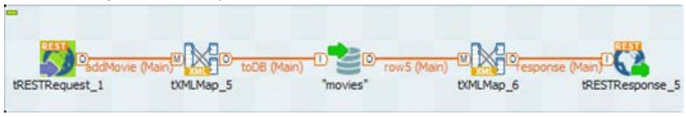
9. From the **Repository**, navigate to the database metadata and place the **movies** table to the right of the **tXMLMap** component. Since you will be writing to the database, choose to use the metadata with a **tMySQLOutput** component.
10. Create a new output from the **tXMLMap**, name it *toDB* and connect it to the **"movies" tMySQLOutput** component. When asked if you want the schema of the target component, click **Yes**. This creates the movies table schema directly in the tXMLMap output flow.
11. To open the editor, double-click the **tXMLMap** component.
12. In the input flow **main:addMovie**, right-click the **body** element, and, on the contextual menu, select **Import From File**. Navigate to the file *C:\StudentFiles\routes\r09_multicast\movie6.xml* and click **OK**.
13. Map the input elements to the output columns. Your **tXMLMap** should appear as in the screenshot below:



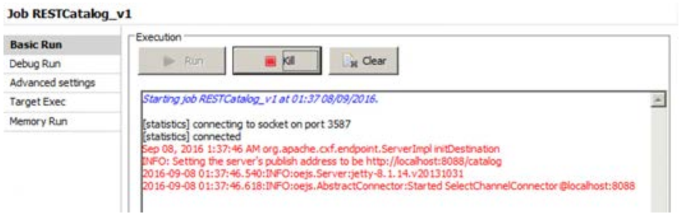
Expression	Column
[addMovie.body:/movie/@ASIN]	movieID
[addMovie.body:/movie/title]	title
[addMovie.body:/movie/director]	director
[addMovie.body:/movie/releaseDate]	releaseDate
[addMovie.body:/movie/runTime]	runtime
[addMovie.body:/movie/studio]	studio
[addMovie.body:/movie/price]	price

To close the **tXMLMap** editor, click **OK**.

14. The last item you need to develop is the response message you must return to the client. Remember that REST services send XML messages. To create the XML response message, add a new **tXMLMap** to the right of the **"movies" tMySQLOutput** component and connect them.
15. Add a **tRESTResponse** component. Create an output from the new **tXMLMap**, name it *response* and link it to the **tRESTResponse** component.



16. Double-click the second **tXMLMap** to build and configure the response message.
17. In the **response** output flow, right-click the **root** element and rename it *message*.
18. In the **Expression** column of the **message** element, enter *"Movie "+row5.movieID+" was inserted in the movie catalog"*. Make sure you replace *row5* with the name of your input row.
19. Run the service.



```
Starting job RESTCatalog_v1 at 01:37 08/09/2016.

[statistics] connecting to socket on port 3587
[statistics] connected
Sep 08, 2016 1:37:46 AM org.apache.cxf.endpoint.ServerImpl initDestination
INFO: Setting the server's publish address to be http://localhost:8088/catalog
2016-09-08 01:37:46.540:INFO:oejs.Server:jetty-8.1.14.v20131031
2016-09-08 01:37:46.618:INFO:oejs.AbstractConnector:Started SelectChannelConnector@localhost:8088
```

Consuming the service

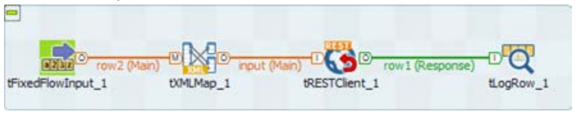
1. Create a new DI Job and name it *RESTConsumer_v3*.
2. Add the following components to the design area:

   tFixedFlowInput—used to generate a single row of data containing movie information

   tXMLMap—used to transform your raw movie data into an XML-structured document tRESTClient—used to send the request to the REST Web service and get a response

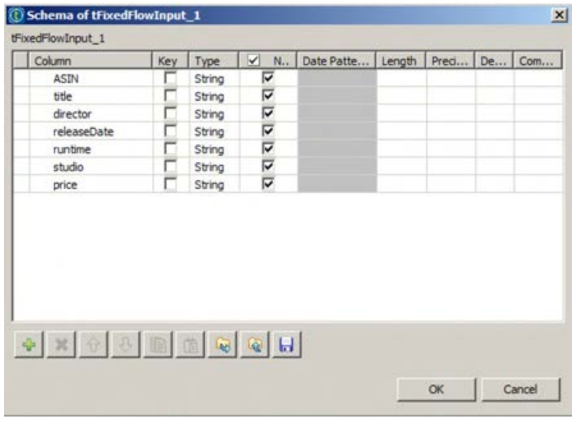
tLogRow—used to display the REST service response on the console

Link the components as shown in the screenshot below.



When linking the **tXMLMap** component to the **tRestClient** component, when asked if you want the schema from the target component, choose **Yes**.

3. In the **tFixedFlowInput Component** view, next to **Edit schema**, click the **Ellipsis (...)** button. In the **Schema editor** window, create the following schema:



tFixedFlowInput_1

Column	Key	Type	N...	Date Patte...	Length	Preci...	De...	Com...
ASIN	☐	String	☑					
title	☐	String	☑					
director	☐	String	☑					
releaseDate	☐	String	☑					
runtime	☐	String	☑					
studio	☐	String	☑					
price	☐	String	☑					

To save, click **OK**.

4. In the **Mode** section, choose **Use Single Table** and enter the following values:

    ASIN: *"B01FJ4UGF0"*

    title: *"Finding Dory"*

    director: *"Andrew Stanton"*

    releaseDate: *"11/15/2016"*

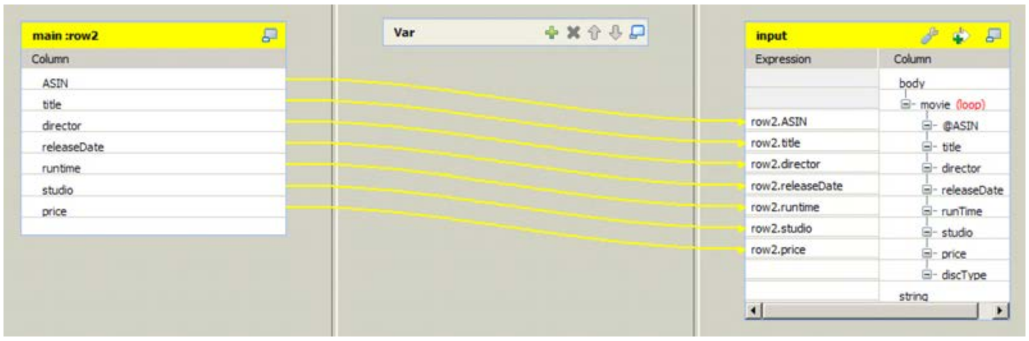
    runtime: *"97"*

    studio: *"Walt Disney"*

    price: *"24.97"*

5. Double-click the **tXMLMap** component. In the output flow, right-click the **body** element, and, on the contextual menu, choose **Import From File**. Navigate to *C:\StudentFiles\routes\r09_multicast*, select **movie6.xml**, and click **OK**.

6. Map the elements as shown in the screenshot below:



To save, click **OK**.

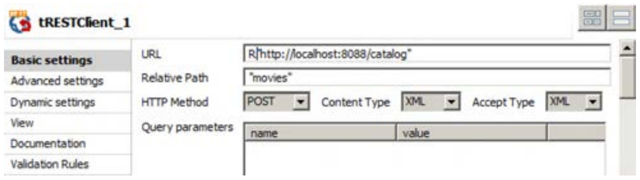
7. In the **tRESTClient Component** view, set the parameters as follows:

   URL: *"http://localhost:8088/catalog"*

   Relative Path: *"movies"*

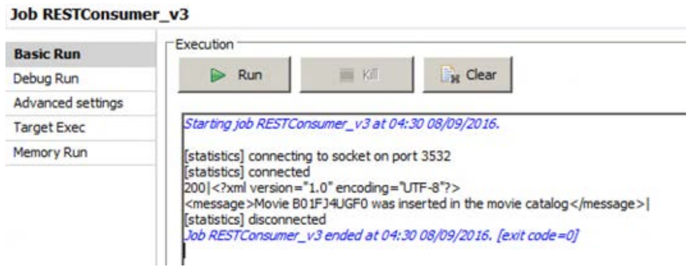
   HTTP Method: *POST*

   Content Type: *XML*



URL: "http://localhost:8088/catalog"
Relative Path: "movies"
HTTP Method: POST Content Type: XML Accept Type: XML

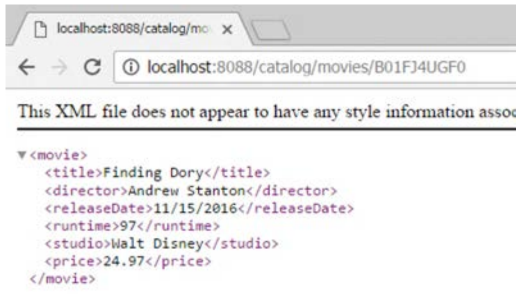
8. Run the Job.



```
Starting job RESTConsumer_v3 at 04:30 08/09/2016.

[statistics] connecting to socket on port 3532
[statistics] connected
200|<?xml version="1.0" encoding="UTF-8"?>
<message>Movie B01FJ4UGF0 was inserted in the movie catalog</message>|
[statistics] disconnected
Job RESTConsumer_v3 ended at 04:30 08/09/2016. [exit code=0]
```

9. To make sure the movie was inserted in the database, request your REST service:


localhost:8088/catalog/mo
localhost:8088/catalog/movies/B01FJ4UGF0

This XML file does not appear to have any style information assoc

```
▼<movie>
   <title>Finding Dory</title>
   <director>Andrew Stanton</director>
   <releaseDate>11/15/2016</releaseDate>
   <runtime>97</runtime>
   <studio>Walt Disney</studio>
   <price>24.97</price>
 </movie>
```

Next step

You have almost completed this lesson. Continue to the Wrap-up section for a review of the concepts we covered.

Wrap-up

REST Web services are considered newer and more lightweight than SOAP Web services. The RESTful API looks familiar to most developers since the operations are standard HTTP methods. REST Web services do not require a contract between the provider and consumer. A REST response replies to the server with single or multiple items using the XML or JSON format.

The tRESTRequest and tRESTResponse components can be used together to provide the results of a Job in response to a Web service provider. This allows a Talend Job to be wrapped as a Web service: a service request is the input for the Job, and the Job result is the service response.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.