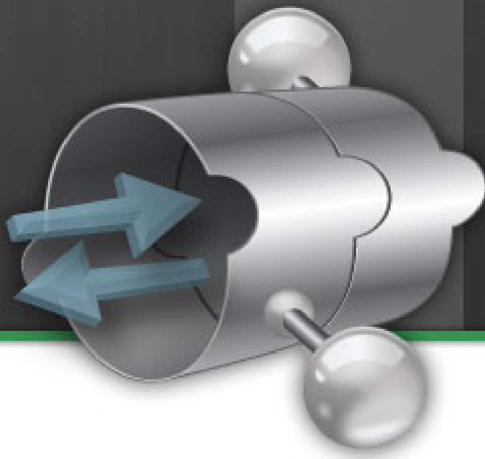


Talend Open Studio

for ESB

Mediation Components Reference Guide

5.1_a

Talend Open Studio : Mediation Components Reference Guide

Adapted for Talend Open Studio for ESB v5.1.x. Supersedes previous Reference Guide releases.

Copyleft

Notices

Table of Contents

Preface

General information

Purpose

This Reference Guide explains in detail the major Camel components of the **Mediation** perspective of *Talend Open Studio for ESB.*

Information presented in this document applies to *Talend Open Studio for ESB* releases beginning with **5.1.x**.

Audience

This guide is for users and administrators of *Talend Open Studio for ESB.*

💡 The layout of GUI screens provided in this document may vary slightly from your actual GUI.

Typographical conventions

This guide uses the following typographical conventions:

- text in **bold:** window and dialog box buttons and fields, keyboard keys, menus, and menu options,
- text in **[bold]:** window, wizard, and dialog box titles,
- text in `courier`: system parameters typed in by the user,
- text in *italics*: file, schema, column, row, and variable names referred to in all use cases, and also names of the fields in the Basic and Advanced setting views referred to in the property table for each component,
- The 💡 icon indicates an item that provides additional information about an important point. It is also used to add comments related to a table or a figure,
- The ⚠ icon indicates a message that gives information about the execution requirements or recommendation type. It is also used to refer to situations or information the end-user need to be aware of or pay special attention to.

History of changes

The following table lists changes made in the *Talend Open Studio for ESB Mediation Components Reference Guide*.

Version	Date	History of Change
v5.0_b	13/02/2012	Separated the appendix for Camel components from *Talend Open Studio for ESB User Guide* to form a new *Talend Open Studio for ESB Mediation Components Reference Guide*.
v5.1_a	03/05/2012	Updates in *Talend Open Studio for ESB Mediation Components Reference Guide* include: • Updated the properties tables and scenarios of some components to match the modifications in the GUI. • Added new components in the Messaging family: cMail and cHttp. • Added a new component in the Context family: cJMSConnectionFactory. • Added a new component in the Miscellaneous family: cLog. • Added a new component in the Exception family: cErrorHandler. • Added a scenario for the cJMS component to explain how to set up a local JMS transaction.

Feedback and Support

Your feedback is valuable. Do not hesitate to give your input, make suggestions or requests regarding this documentation or product and find support from the **Talend** team, on **Talend**'s Forum website at:

http://talendforge.org/forum

Context components

This chapter details the major components that you can find in the **Context** family from the **Palette** of the **Mediation** perspective of *Talend Open Studio for ESB*.

The **Context** family groups components that define contexts you want to use in your Routes.

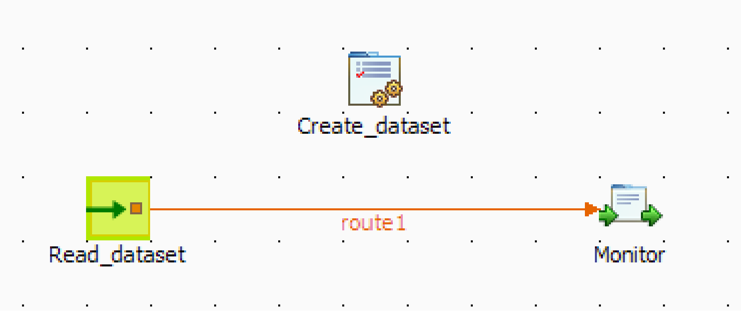
cConfig

cConfig properties

Component Family	Context	
Function	**cConfig** allows you to set the CamelContext.	
Purpose	**cConfig** manipulates the Camel context as needed by the Routes.	
Basic settings	*Code*	Write a piece of code to manipulate the CamelContext.
	Dependencies	Select the library or libraries that is required by the CamelContext or Typeconverter Registry from the list.
Usage	**cConfig** cannot be added directly in a Route.	
Limitation	n/a	

Scenario: Implementing a dataset from the Registry

In this scenario, an instance of dataset is added in the Registry and implemented by a **cMessagingEndpoint** component.



Dropping and linking the components

1. From the **Palette**, expand the **Context** folder, and drop a **cConfig** component onto the design workspace.

2. Expand the **Messaging** folder, and drop a **cMessagingEndpoint** component onto the design workspace.

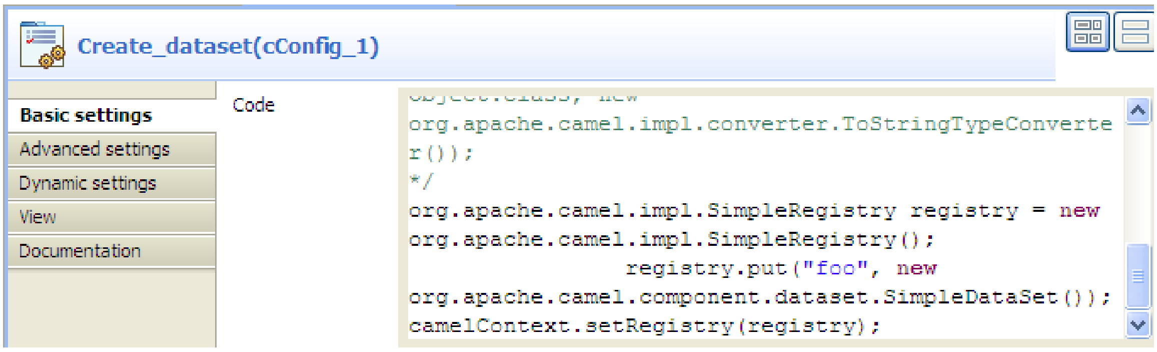
3. Expand the **Processor** folder, and drop a **cProcessor** component onto the design workspace.

4. Right-click the input **cMessagingEndpoint** component, select **Row** > **Route** from the contextual menu and click the **cProcessor** component.

5. Label the components to better identify their functionality.

Configuring the components

1. Double-click the **cConfig** component, which is labelled *Create_dataset*, to display its **Basic settings** view in the **Component** tab. and set its parameters.



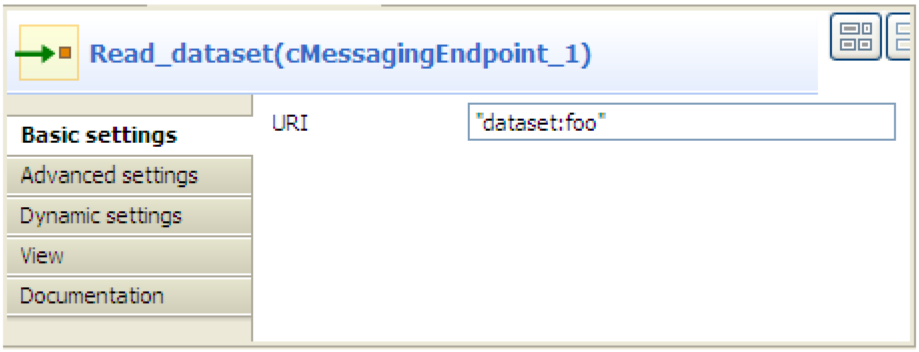
Code

```
org.apache.camel.impl.converter.ToStringTypeConverte
r());
*/
org.apache.camel.impl.SimpleRegistry registry = new
org.apache.camel.impl.SimpleRegistry();
                registry.put("foo", new
org.apache.camel.component.dataset.SimpleDataSet());
camelContext.setRegistry(registry);
```

2. Write a piece of code in the **Code** field to register the dataset instance *foo* into the registry, as shown below.

```
org.apache.camel.impl.SimpleRegistry registry = new
org.apache.camel.impl.SimpleRegistry();
                registry.put("foo", new
org.apache.camel.component.dataset.SimpleDataSet());
camelContext.setRegistry(registry);
```

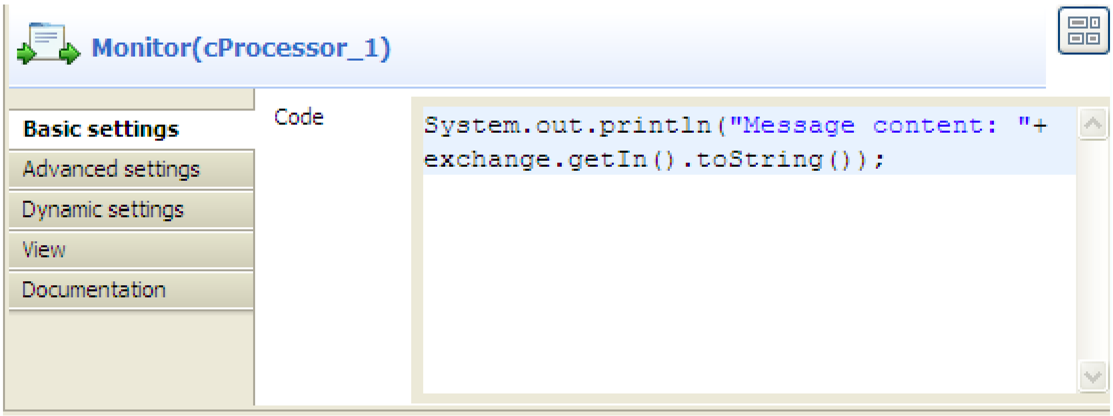
3. Double-click the input **cMessagingEndpoint** component, which is labelled *Read_dataset*, to display its **Basic settings** view in the **Component** tab.



URI "dataset:foo"

4. In the **URI** field, enter *dataset:foo* between the quotation marks.

5. Double-click the **cProcessor** component, which is labelled *Monitor*, to display its **Basic settings** view in the **Component** tab.



Code

```
System.out.println("Message content: "+
exchange.getIn().toString());
```

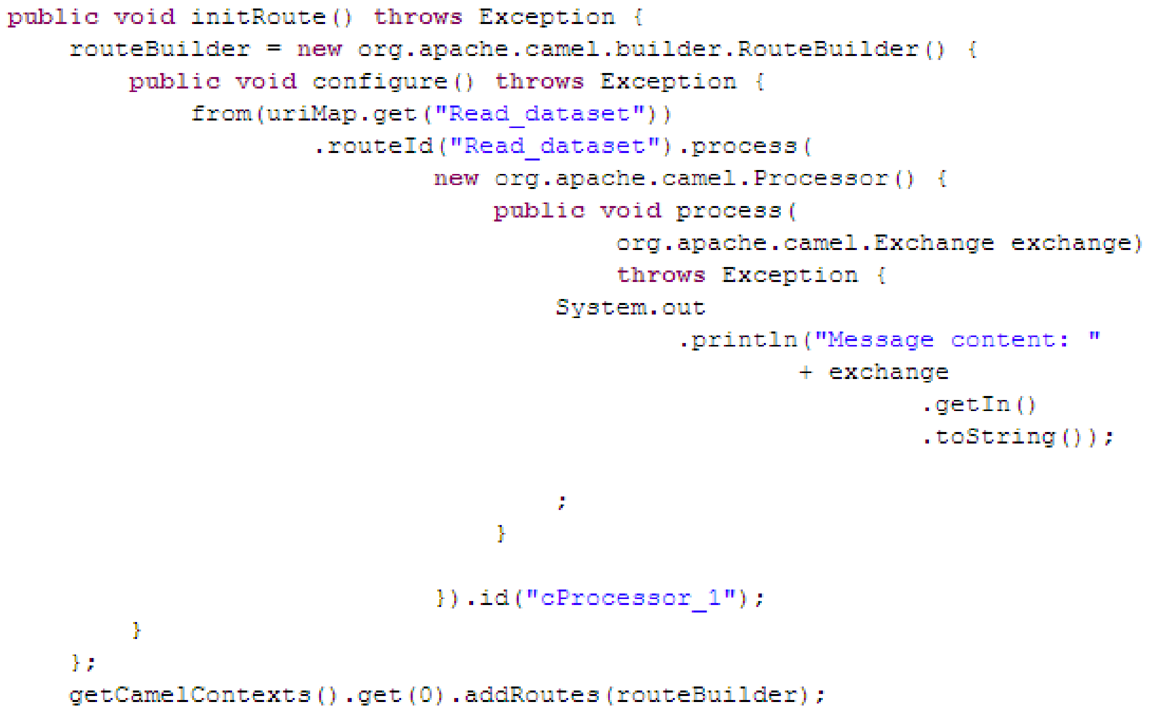
6. In the **Code** box, customize the code as follows so that the **Run** console displays the message contents:

```
System.out.println("Message content: "+
exchange.getIn().toString());
```

7. Press **Ctrl+S** to save your route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

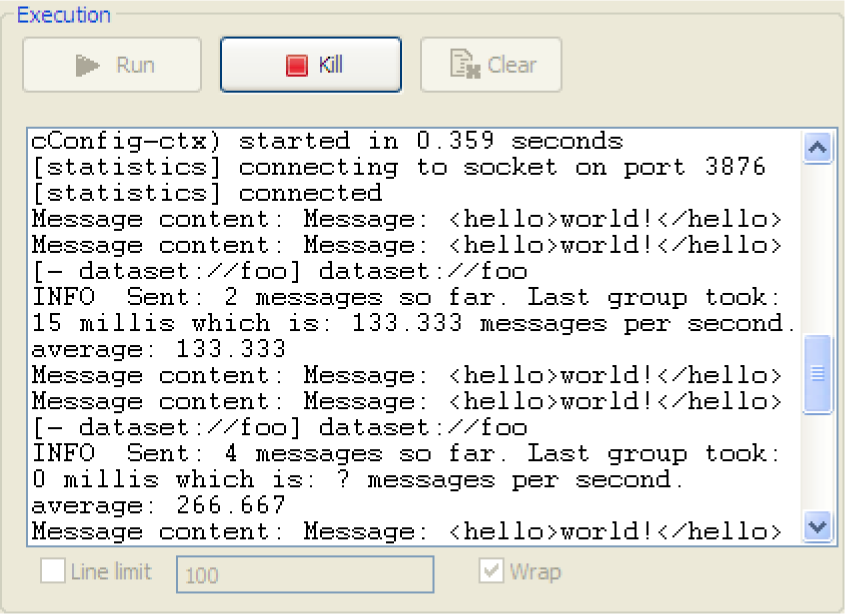


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Read_dataset"))
                    .routeId("Read_dataset").process(
                            new org.apache.camel.Processor() {
                                public void process(
                                        org.apache.camel.Exchange exchange)
                                        throws Exception {
                                    System.out
                                            .println("Message content: "
                                                    + exchange
                                                            .getIn()
                                                            .toString());

                                    ;
                                }

                            }).id("cProcessor_1");
        }
    };
    getCamelContexts().get(0).addRoutes(routeBuilder);
```

As shown in the code, a message route is built `from` the endpoint identified by`Read_dataset` and `cProcessor_1` gets the message content and displays it on the console.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your route. You can also press **F6** to execute it.

   RESULT: The message content is printed in the console.



```
cConfig-ctx) started in 0.359 seconds
[statistics] connecting to socket on port 3876
[statistics] connected
Message content: Message: <hello>world!</hello>
Message content: Message: <hello>world!</hello>
[- dataset://foo] dataset://foo
INFO  Sent: 2 messages so far. Last group took:
15 millis which is: 133.333 messages per second.
average: 133.333
Message content: Message: <hello>world!</hello>
Message content: Message: <hello>world!</hello>
[- dataset://foo] dataset://foo
INFO  Sent: 4 messages so far. Last group took:
0 millis which is: ? messages per second.
average: 266.667
Message content: Message: <hello>world!</hello>
```

cJMSConnectionFactory

cJMSConnectionFactory properties

Component Family	Context	
Function	**cJMSConnectionFactory** specifies the connection factory that can be used by multiple **cJMS** components in a Route.	
Purpose	**cJMSConnectionFactory** is used to specify the JMS connection factory for message handling.	
	MQ Server	Select an MQ server from **ActiveMQ**, **Customized**, or **WebSphere MQ**.
	Use Transaction	Select this check box to enable local transaction in the current Route.
	Broker URI (for **ActiveMQ** only)	Type in the URI of the message broker. For intra-Route message handling, you can simply use the default URI *vm://localhost?broker.persistent=false*.
	Use PooledConnectionFactory (for **ActiveMQ** only)	Select this check box to use PooledConnectionFactory.
	Max Connections (for **ActiveMQ** only)	Specify the maximum number of connections of the PooledConnectionFactory. This field is available only when the **Use PooledConnectionFactory** check box is selected.
	Max Active (for **ActiveMQ** only)	Specify the maximum number of sessions per connection. This field is available only when the **Use PooledConnectionFactory** check box is selected.
	Idle Timeout (for **ActiveMQ** only)	Specify the maximum waiting time before the connection breaks. This field is available only when the **Use PooledConnectionFactory** check box is selected.
	Expiry Timeout (for **ActiveMQ** only)	Specify the time before the connection breaks since it is used for the first time. This field is available only when the **Use PooledConnectionFactory** check box is selected.
	Codes (for **Customized** only)	Write a piece of code to specify the JMS connection factory to be used for message handling.
	Dependencies (for **Customized** only)	Specify the library or libraries required by the JMS connection factory.
	Host Name (for **WebSphere MQ** only)	Type in the name or IP address of the host on which the IBM WebSphere MQ server is running.
	Port (for **WebSphere MQ** only)	Type in the port of the IBM WebSphere MQ server, **1414** by default.

	Transport Type (for **WebSphere MQ** only)	Select a type of message transport between the IBM WebSphere MQ server and the WebSphere MQ broker from **Bindings**, **Bindings then Client**, and **Client**.
	Queue Manager (for **WebSphere MQ** only)	Type in the name of the queue manager, or specify the name of the IBM WebSphere MQ server to find a queue manager.
	Authentication (for **WebSphere MQ** only)	On some operating systems, select this check box and provide the username and password for the IBM WebSphere MQ server to validate the access permission. This option is not required on Windows.
	Dependencies (for **WebSphere MQ**)	Specify additional libraries required by the IBM WebSphere MQ broker, which are normally provided with the server installer.
Usage	**cJMSConnectionFactory** cannot be added directly in a Route.	
Limitation	n/a	

Related scenario:

For a related scenario, see the section called “Scenario: Sending and receiving a message from a JMS queue”.

Exception components

This chapter details the major components that you can find in the **Exception** family from the **Palette** of the **Mediation** perspective of *Talend Open Studio for ESB*.

The **Exception** family groups components that are dedicated to exception handling of Routes.

cErrorHandler

cErrorHandler properties

Component Family	Exception	
Function	**cErrorHandler** provides multiple strategies to deal with errors processing an Event Driven Consumer.	
Purpose	**cErrorHandler** offers different strategies for error handling during the processing.	
Basic settings	*Default Handler*	This error handler does not support a dead letter queue and will return exceptions back to the caller.
		Set Maximum Redeliveries: select this check box to set the number of redeliveries in the **Maximum Redeliveries (int)** field.
		Set Redelivery Delay: select this check box to set the initial redelivery delay (in milliseconds) in the **Redelivery Delay (long)** field.
		Set Retry Attempted Log Level: select this check box to select the log level in the **Level** list for log messages when retries are attempted.
		Asynchronized Delayed Redelivery: select this check box to allow asynchronous delayed redelivery.
		Use Original Message: select this check box to use the original message for redelivery.
		More Configurations by Code: select this check box to enter codes in the **Code** box for further configuration.
	Dead Letter	This handler supports attempting to redeliver the message exchange a number of times before sending it to a dead letter endpoint.
		Dead Letter Uri: select this check box to define the endpoint of the dead letter queue. Other parameters share the same meaning as those of the default handler.
	Logging Handler	This handler logs the exceptions.
		Set Logger Name: select this check box to give a name to the logger in the **Name** field.
		Set Log Level: select this check box to decide the log level from the **Level** list.
Usage	**cErrorHandler** provides multiple strategies to deal with errors processing an Event Driven Consumer.	
Limitation		

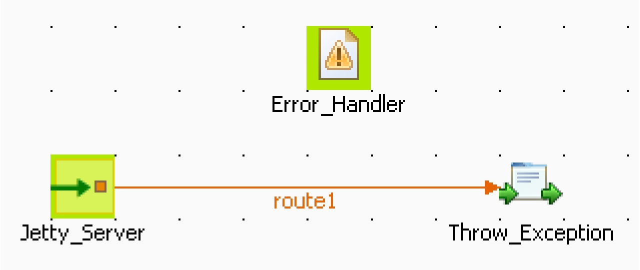
Scenario: Logging the exception thrown during a client/server talk

In this scenario, a Jetty server is started before a client browser requests access to it. Then an exception is thrown at the server side and logged by **cErrorHandler**.

Dropping and linking the components

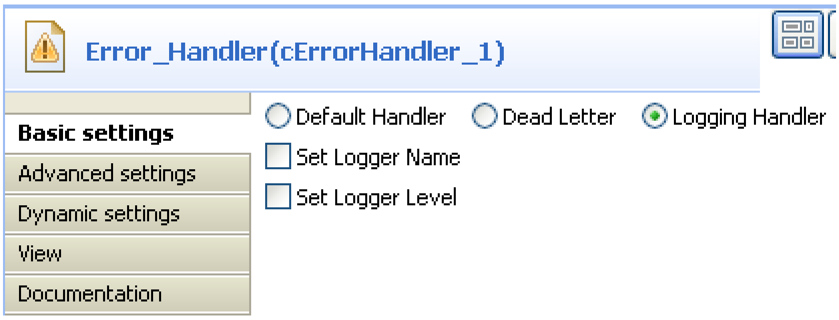
1. Drop the following components from the **Palette** onto the workspace: **cMessagingEndpoint**, **cErrorHandler** and **cProcessor**, labelled as **Jetty_Server**, **Error_Handler** and **Throw_Exception** respectively.

2. Link **cMessagingEndpoint** and **cProcessor** using a **Row** > **Route** connection.



Configuring the components

1. Double-click **cErrorHandler** to open its **Basic settings** view in the **Component** tab.



Error_Handler(cErrorHandler_1)

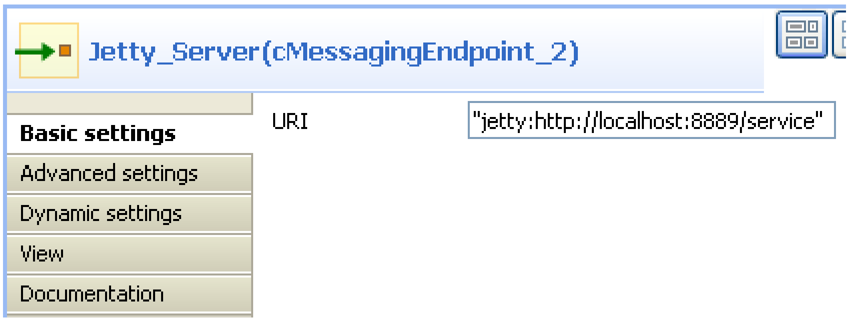
○ Default Handler ○ Dead Letter ● Logging Handler

□ Set Logger Name

□ Set Logger Level

2. Select **Logging Handler** to log the exceptions that are thrown.

3. Double-click **cMessagingEndpoint** to open its **Basic settings** view in the **Component** tab.

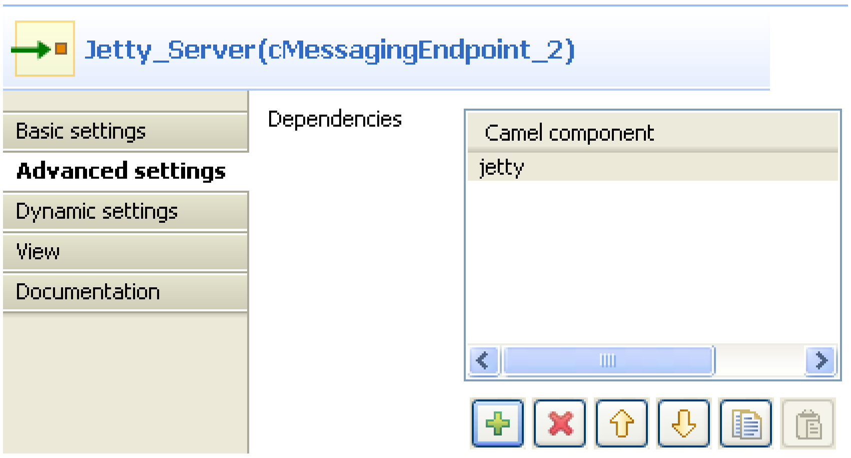


Jetty_Server(cMessagingEndpoint_2)

URI "jetty:http://localhost:8889/service"

4. In the **Uri** field, enter `jetty:http://localhost:8889/service` to specify the Jetty server.

5. Click **Advanced settings** for further setup.

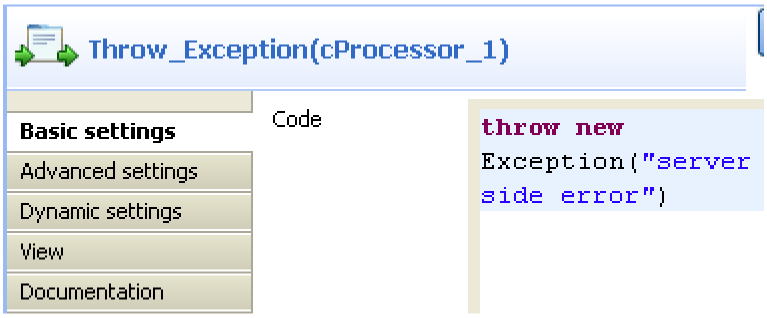


Dependencies

Camel component
jetty

6. In the **Dependencies** table, click the **[+]** button to add a line and select `jetty` from the **Camel component** list.

7. Double-click **cProcessor** to open its **Basic settings** view in the **Component** tab.



Code

```
throw new
Exception("server
side error")
```

8. In the **Code** box, enter `throw new Exception("server side error")` to throw an exception.

9. Press **Ctrl+S** to save your Route.

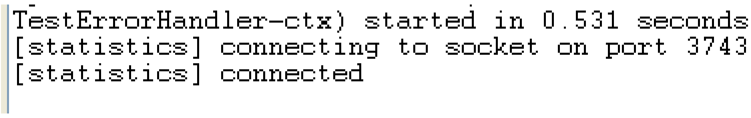
Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.

```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            errorHandler(loggingErrorHandler())

            ;
            from(uriMap.get("Jetty_Server"))
                    .routeId("Jetty_Server").process(
                            new org.apache.camel.Processor() {
                                public void process(
                                        org.apache.camel.Exchange exchange)
                                        throws Exception {
                                    throw new Exception(
                                            "server side error");
                                }

                            }).id("cProcessor_1");
```

As shown above, the route starts `from` the endpoint `Jetty_Server` and throws the exception of `server side error` via `cProcessor_1`.

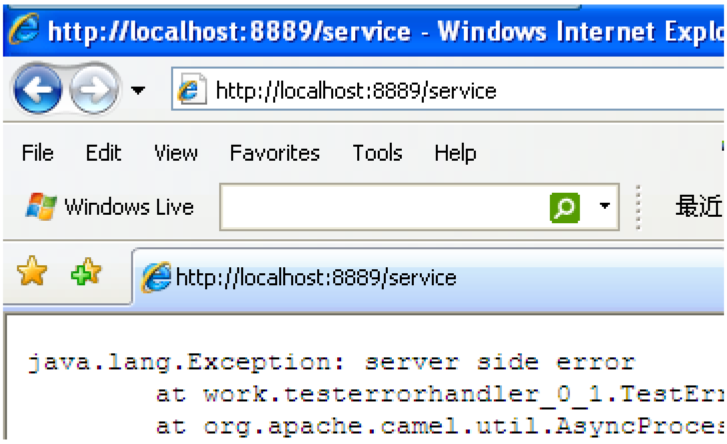
2. Press **F6** to execute the Route.



```
TestErrorHandler-ctx) started in 0.531 seconds
[statistics] connecting to socket on port 3743
[statistics] connected
```

The Jetty server has started.

3. Launch an Internet browser and enter `http://localhost:8889/service` (the Jetty server URI configured above) in the address bar to access the server.

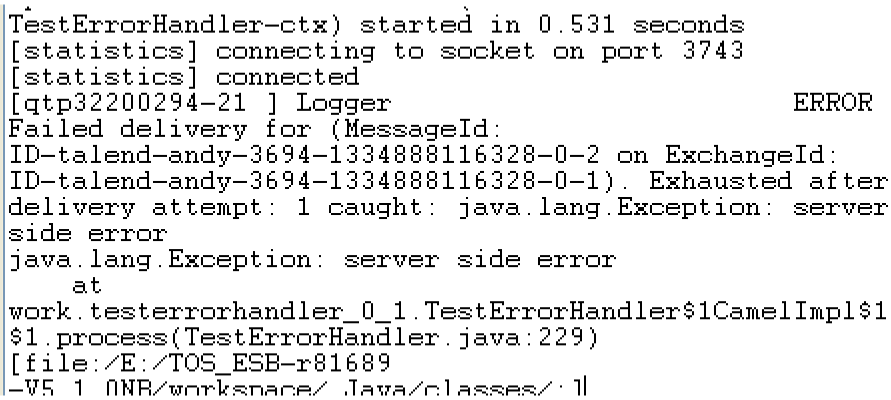


http://localhost:8889/service - Windows Internet Explo
http://localhost:8889/service
File
Edit
View
Favorites
Tools
Help
Windows Live
最近
http://localhost:8889/service

```
java.lang.Exception: server side error
        at work.testerrorhandler_0_1.TestErr
        at org.apache.camel.util.AsyncProces
```

As shown above, the request failed due to the server error.

4. Go to the Studio and check the results in the **Run** tab.



```
TestErrorHandler-ctx) started in 0.531 seconds
[statistics] connecting to socket on port 3743
[statistics] connected
[qtp32200294-21 ] Logger                         ERROR
Failed delivery for (MessageId:
ID-talend-andy-3694-1334888116328-0-2 on ExchangeId:
ID-talend-andy-3694-1334888116328-0-1). Exhausted after
delivery attempt: 1 caught: java.lang.Exception: server
side error
java.lang.Exception: server side error
    at
work.testerrorhandler_0_1.TestErrorHandler$1CamelImpl$1
$1.process(TestErrorHandler.java:229)
[file:/E:/TOS_ESB-r81689
-V5_1_0NB/workspace/.Java/classes/:]
```

As shown above, **cErrorHandler** has logged the exception at the level of *ERROR*.

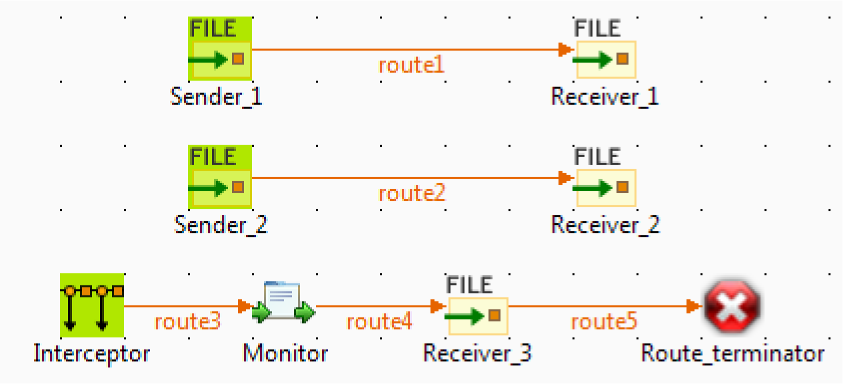
cIntercept

cIntercept properties

Component Family	Exception	
Function	**cIntercept** intercepts the messages in all the sub-routes on a Route before they are produced, and routes them in a new single sub-route without modifying the original ones. When this detour is complete, message routing to the originally intended target endpoints continues.	
Purpose	**cIntercept** intercepts each message sub-route and redirects it in another sub-route without modifying the original one. This can be useful at testing time to simulate error handling.	
Usage	**cIntercept** is a start component of a sub-route.	
Connections	*Row / Route*	Select the **Route** link to intercept all the messages of all the sub-routes listened to by the **cIntercept**.
	Trigger / When	Select the **When** link to filter the messages to intercept and click the **Component** view. In the **Type** list, select the type of language you will use to declare your condition. In the **Condition** field, type in the condition that will be used to filter the messages. All the messages that do not match this condition are dropped by default or can be retrieved with the **Otherwise** link to a different channel.
Limitation	To keep the original sub-routes untouched, cIntercept only be used in a separate sub-route .	

Scenario: Intercepting several routes and redirect them in a single new route

In this scenario, messages on two sub-routes are intercepted and routed along a new sub-route, which is then terminated before the original sub-routes continue.



Dropping and linking the components

This scenario requires five **cFile** components, one **cIntercept** component, one **cProcessor** component, and one **cStop** component.

1. From the **Messaging** folder of the **Palette**, drop four **cFile** components onto the design workspace.

2. Connect the two pairs of **cFile** components using **Row** > **Route** connections. Messages on these two sub-routes will be intercepted.

3. From the **Exception** folder, drop a **cIntercept** component onto the design workspace.

4. From the **Processor** folder, drop a **cProcessor** component onto the design workspace.

5. From the **Messaging** folder, drop a fifth **cFile** component onto the design workspace.

6. From the **Miscellaneous** folder, drop a **cStop** component onto the design workspace.

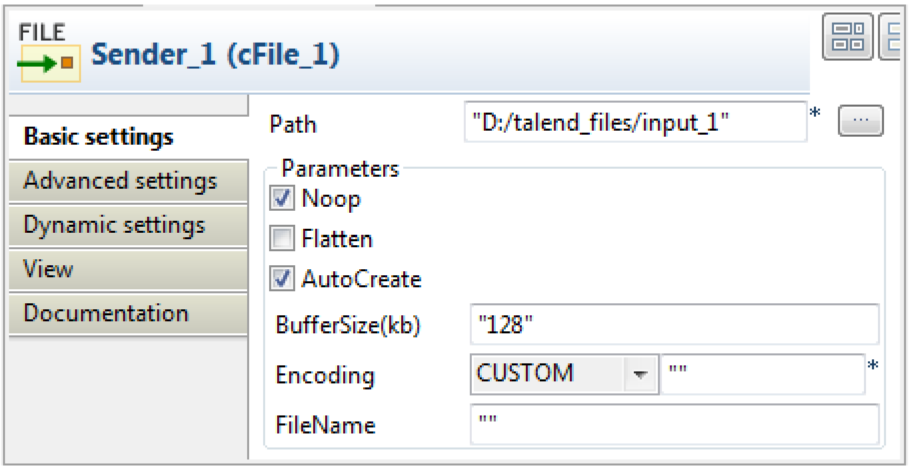
7. Connect these four components one to the next using **Row** > **Route** connections. Along this sub-route, intercepted messages will be directed to a new endpoint before the entire Route is terminated.

8. Label the components to better identify their roles in the Route.

Configuring the components and connections

In this scenario, the **cIntercept** component intercepts all the messages on all the sub-routes as soon as the messages are sent and does not have properties to set. The **cStop** component stops the sub-route on which it is dropped before it completes and does not have properties to set. Therefore, you only need to configure the messaging endpoints and monitor components.

1. Double-click the **cFile** component labeled *Sender_1* to display its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/input_1" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

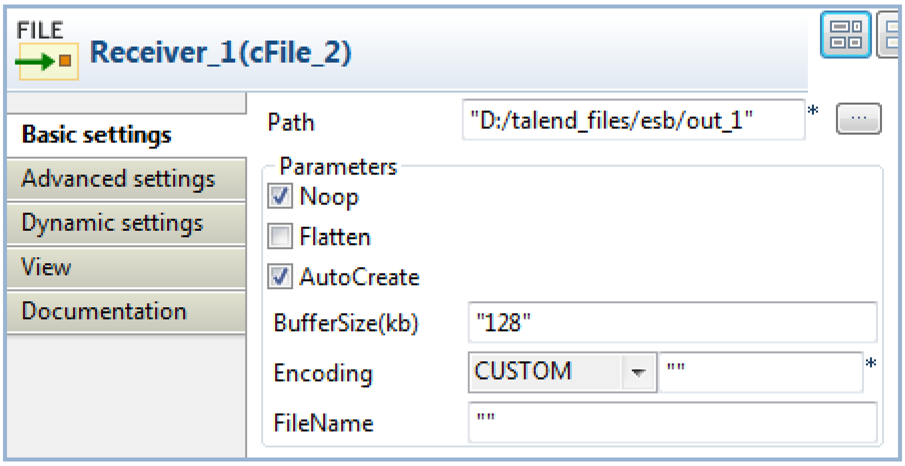
BufferSize(kb) "128"

Encoding CUSTOM "" *

FileName ""

2. In the **Path** field, specify the file path to the first source your are going to send messages from, and leave the other parameters as they are.

3. Double-click the **cFile** component labeled *Receiver_1* to display its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/esb/out_1" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

Encoding CUSTOM "" *

FileName ""

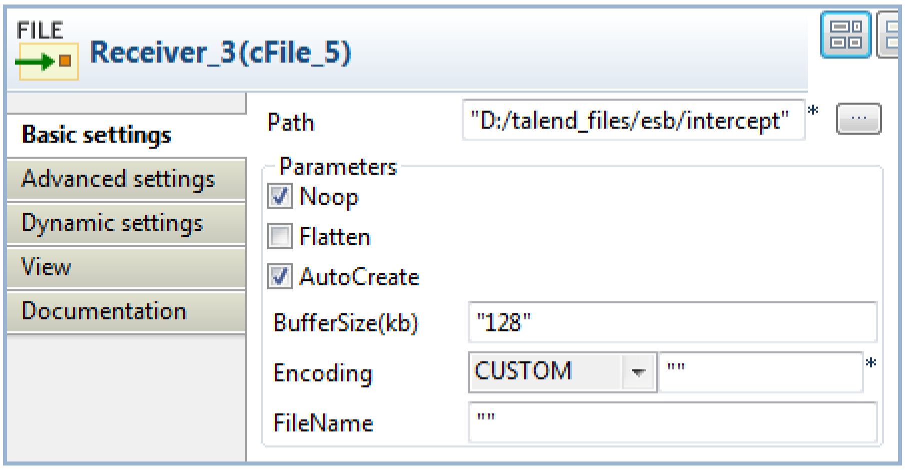
4. In the **Path** field, specify the file path to the first destination you are going to send messages to, and leave the other parameters as they are.

5. In the same way, set the **cFile** components labeled *Sender_2* and *Receiver_2* across the second sub-route.

6. Double-click the **cProcessor** component, which is labeled *Monitor*, to display its **Basic settings** view in the **Component** tab, and customize the code in the **Code** area to display the file names of the messages intercepted on the console:

```
System.out.println("Message intercepted: "+
exchange.getIn().getHeader("CamelFileName"));
```

7. Double-click the **cFile** component labeled *Receiver_3* to display its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/esb/intercept"

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

Encoding CUSTOM ""

FileName ""

8. In the **Path** field, specify the file path to the destination for the intercepted messages, and leave the other parameters as they are.

9. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

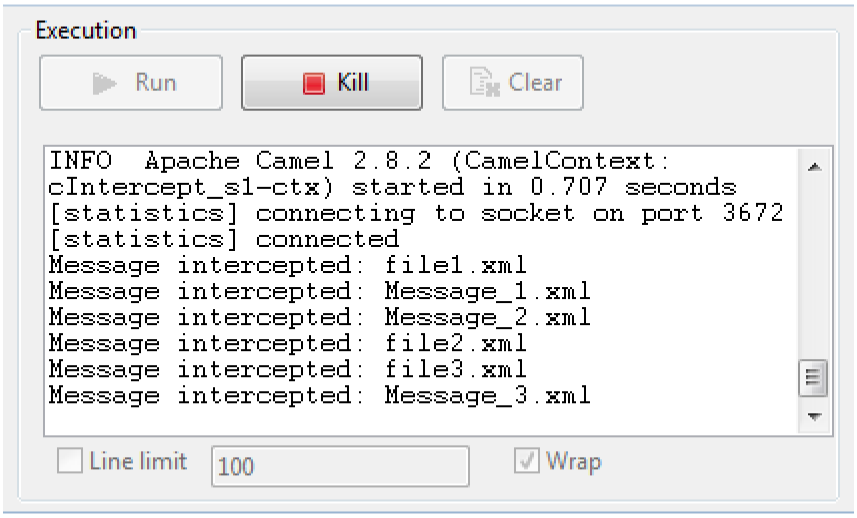
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            intercept()
                    .routeId("Interceptor")
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out
                                    .println("Message intercepted: "
                                            + exchange
                                                    .getIn()
                                                    .getHeader(
                                                            "CamelFileName"));
                            ;
                        }

                    }).id("cProcessor_1").to(
                            uriMap.get("Receiver_3")).id("cFile_5")
                    .stop()
                    .id("cStop_1");
            from(uriMap.get("Sender_1 ")).routeId("Sender_1 ").to(
                    uriMap.get("Receiver_1")).id("cFile_2");
            from(uriMap.get("Sender_2")).routeId("Sender_2").to(
                    uriMap.get("Receiver_2")).id("cFile_4");
        }
    };
    getCamelContexts().get(0).addRoutes(routeBuilder);
```

As shown in this piece of code, *Interceptor* intercepts all messages on route, the intercepted messages are directed `.to` the endpoint *Receiver_3*, and *cStop_1* terminates message routing before the messages are routed `from` the endpoint *Sender_1* `.to` the endpoint *Receiver_1* and `from` the endpoint *Sender_2* `.to` the endpoint *Receiver_2*.

2. Click the **Run** view and click the **Run** button to launch the execution of your Route. You can also press **F6** to execute it.

   RESULT: Files are sent from the endpoints, caught by the **cIntercept** component, monitored by the **cProcessor** component and sent to a new endpoint, and then the original sub-routes are terminated before they can continue.



```
INFO  Apache Camel 2.8.2 (CamelContext:
cIntercept_s1-ctx) started in 0.707 seconds
[statistics] connecting to socket on port 3672
[statistics] connected
Message intercepted: file1.xml
Message intercepted: Message_1.xml
Message intercepted: Message_2.xml
Message intercepted: file2.xml
Message intercepted: file3.xml
Message intercepted: Message_3.xml
```

□ Line limit 100 ☑ Wrap

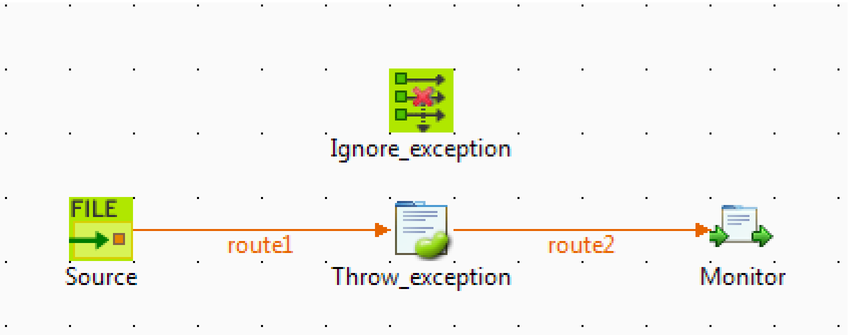
cOnException

cOnException properties

Component Family	Exception	
Function	**cOnException** catches the defined exceptions to trigger desired actions.	
Purpose	**cOnException** is designed to catch the defined exceptions for desired error handling.	
Basic settings	*Exceptions*	Click the plus button to add as many lines as needed in the table to define the exceptions to be caught.
	Set a redelivering tries count	Select this check box to set the maximum redelivering tries in the **Maximum redelivering tries** field.
	Non blocking asynchronous behavior	Select this check box to enable the feature of not blocking asynchronous behavior.
	Exception behavior	**None**: select this option to take no action on the original route. **Handle the exceptions**: select this option to handle exceptions and break out the original route. **Ignore the exceptions**: select this option to ignore the exceptions and continue routing in the original route.
	Route the original input body instead of the current body	Select this check box to route the original message instead of the current message that might be changed during the routing.
Usage	**cOnException** is generally used as a standalone component in a sub-route.	
Limitation	n/a	

Scenario: Using cOnException to ignore exceptions and continue message routing

In this scenario, a **cOnException** component is used to ignore an IO exception thrown by a Java bean so that the message is successfully routed to the destination in spite of the exception.



Dropping and linking the components

1. Drag and drop these components from the **Palette** onto the workspace: a **cOnException** component, a **cFile** component, a **cBean** component, and **cProcessor** component.

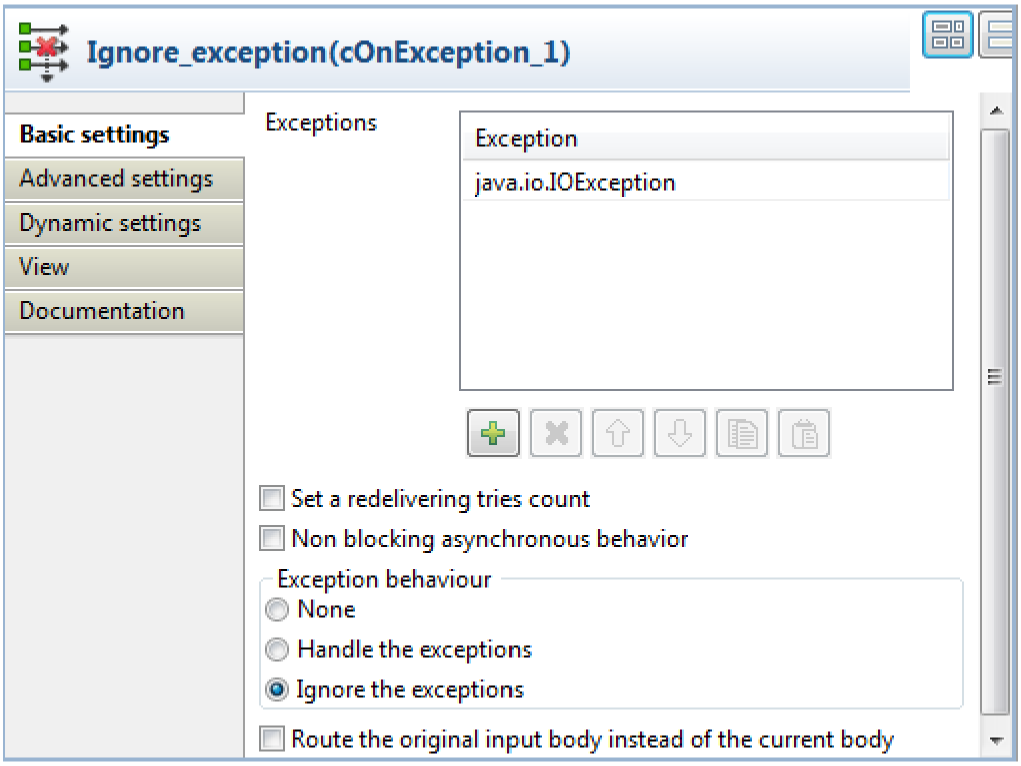
2. Link **cFile** to **cBean** using a **Row** > **Route** connection.

3. Link **cBean** to **cProcessor** using a **Row** > **Route** connection.

4. Label the components to better identify their roles in the Route.

Configuring the components

1. Double-click the **cOnException** component, which is labelled *Ignore_exception*, to open its **Basic settings** view in the **Component** tab.



Ignore_exception(cOnException_1)

Exceptions

Exception
java.io.IOException

□ Set a redelivering tries count

□ Non blocking asynchronous behavior

Exception behaviour

○ None

○ Handle the exceptions

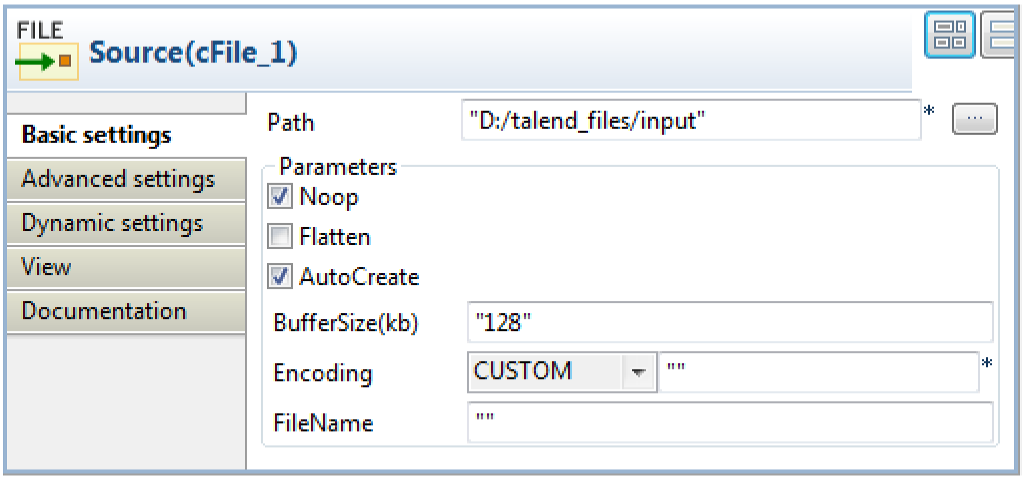
● Ignore the exceptions

□ Route the original input body instead of the current body

2. Click the plus button to add a line in the **Exceptions** table, and define the exception to catch. In this example, enter `java.io.IOException` to handle IO exceptions.

   In the **Exception behavior** area, select the **Ignore the exceptions** option to ignore exceptions and let message routing continue. Leave the other parameters as they are.

3. Double-click the **cFile** component, which is labelled *Source*, to open its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/input" *

Parameters

☑ Noop

☐ Flatten

☑ AutoCreate

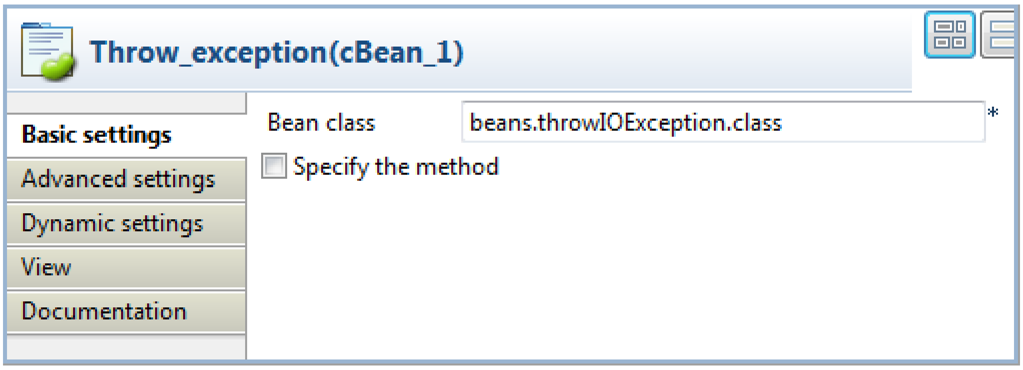
BufferSize(kb) "128"

Encoding CUSTOM "" *

FileName ""

4. In the **Path** field, enter the path of the message source, and leave the other parameters as they are.

5. Double-click the **cBean** component, which is labelled *Throw_exception*, to open its **Basic settings** view in the **Component** tab.



Bean class beans.throwIOException.class *

☐ Specify the method

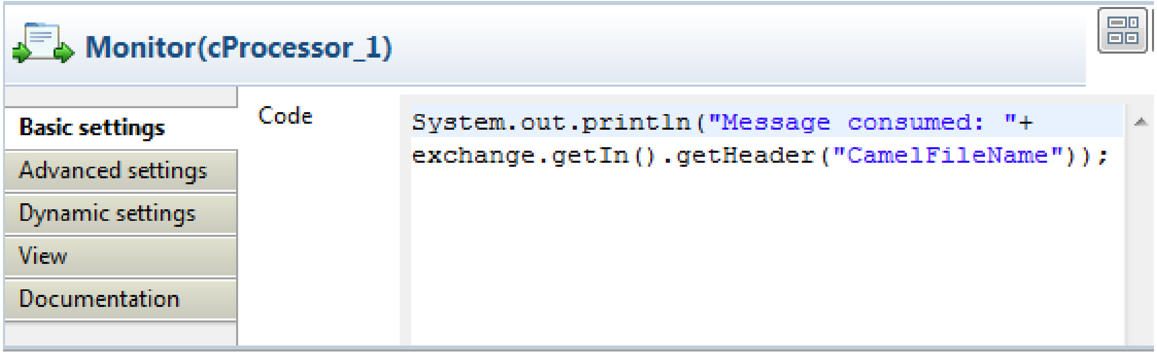
6. In the **Bean class** field, enter the name of the bean to throw an IO exception, *beans.throwIOException.class* in this scenario.

   Note that this bean has already been defined in the **Code** node of the **Repository** and it looks like this:

```
package beans;

import java.io.IOException;

import org.apache.camel.Exchange;


public class throwIOException {

    /**

     * @throws IOException
     */
    public static void helloExample(String message, Exchange exchange)
 throws IOException {
        throw new IOException("An IOException has been caught");
    }
}
```

   For more information about creating and using Java Beans, see *Talend Open Studio for ESB* **User Guide**.

7. Double-click the **cProcessor** component, which is labelled *Monitor*, to open its **Basic settings** view in the **Component** tab.



```
System.out.println("Message consumed: "+
exchange.getIn().getHeader("CamelFileName"));
```

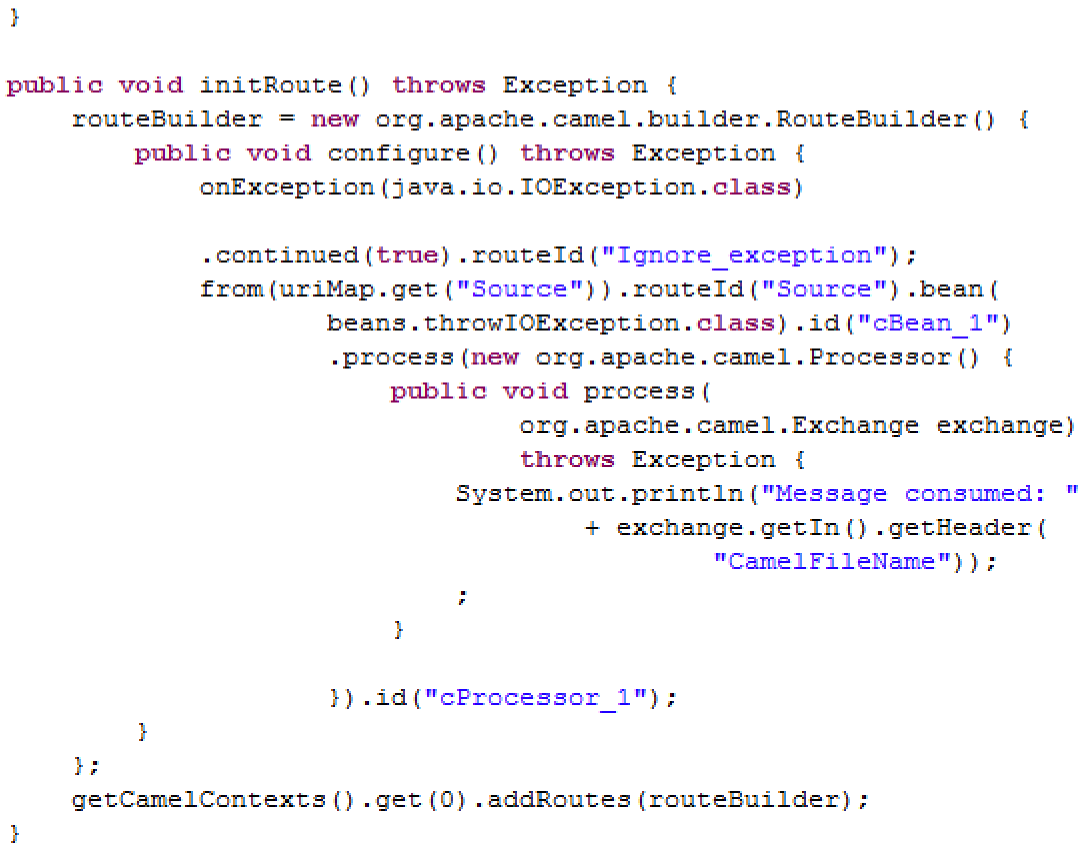
8. In the **Code** area, customize the code to display the file name of the consumed message on the **Run** console:

```
System.out.println("Message consumed: "+
exchange.getIn().getHeader("CamelFileName"));
```

9. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.

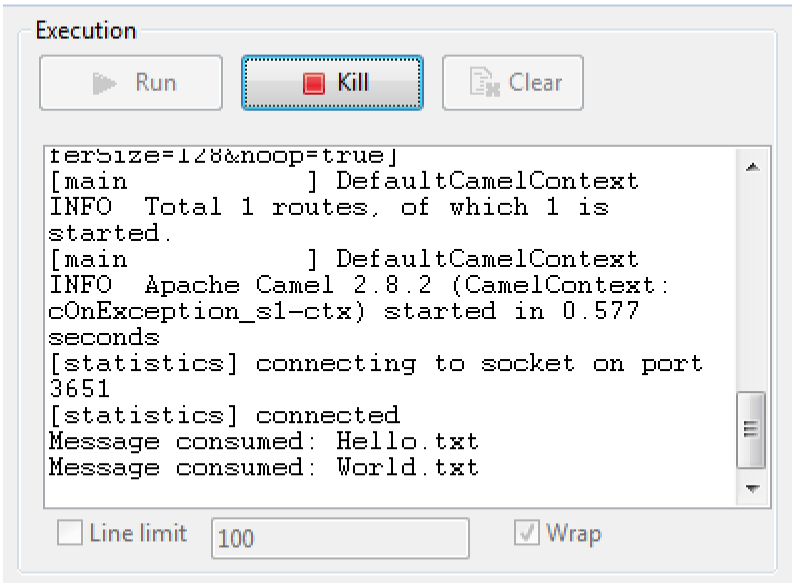


```
}

public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            onException(java.io.IOException.class)

            .continued(true).routeId("Ignore_exception");
            from(uriMap.get("Source")).routeId("Source").bean(
                    beans.throwIOException.class).id("cBean_1")
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out.println("Message consumed: "
                                    + exchange.getIn().getHeader(
                                            "CamelFileName"));
                            ;
                        }

                    }).id("cProcessor_1");
        }
    };
    getCamelContexts().get(0).addRoutes(routeBuilder);
}
```

As shown above, `Ignore_exception` handles any IO exception thrown by `.bean(beans.throwIOException.class)` invoked by `cBean_1`, so that messages `from` the endpoint `Source` can be successfully routed onwards (`continued(true)`) in spite of the exception.

2. Press **F6** to execute the Route.

   The route gets executed successfully and the files from the source are successfully routed to the destination.



```
ferSize=128&noop=true]
[main              ] DefaultCamelContext
INFO  Total 1 routes, of which 1 is
started.
[main              ] DefaultCamelContext
INFO  Apache Camel 2.8.2 (CamelContext:
cOnException_s1-ctx) started in 0.577
seconds
[statistics] connecting to socket on port
3651
[statistics] connected
Message consumed: Hello.txt
Message consumed: World.txt
```

3. Change the exception handling option in the **cOnException** component or deactivate the component and run the Route again.

   The exception thrown by the Java bean prevents the messages from being routed successfully.

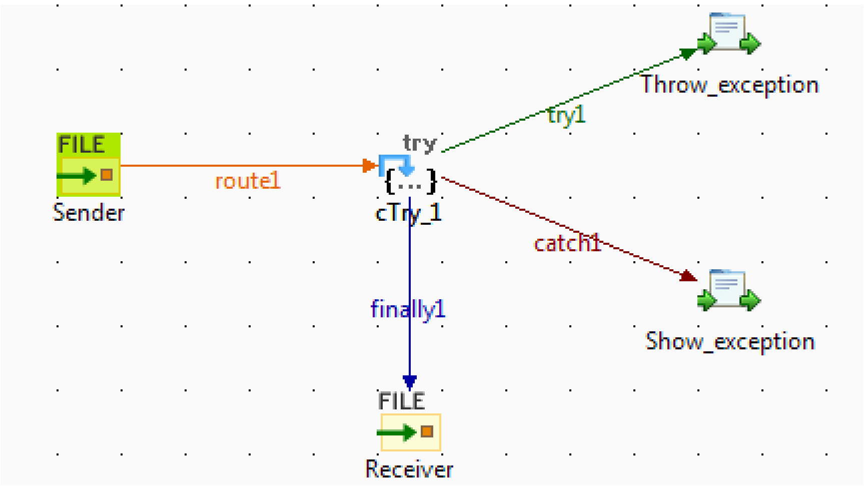
cTry

cTry properties

Component Family	Exception	
Function	**cTry** offers Java's exception handling abilities by building Try/Catch/Finally blocks.	
Purpose	**cTry** is designed to build Try/Catch/Finally blocks to handle exceptions.	
Usage	**cTry** is used as a middle component in a Route.	
Connections	*Try*	Select this link to isolate the part of your Route that is likely to throw an exception or exceptions. When the **Try** link is followed by multiple components, a compile error may occur showing `"The method doCatch() is undefined for the type ExpressionNode"`. In this case, use a **cJavaDSLProcessor** component to end the Try block with the code `.endDoTry()` as a workaround.
	Catch	Select this link to catch any exception thrown in the Route. In the **Exceptions** field, type in an expression to filter the type of exception to catch. This link can be used only when a Try link is present.
	Finally	Select link to execute final instructions regardless of any exceptions that may occur in the Route. This link can be used only when a Try link is present.
	Route	Select this link to route all the messages from the sender to the next endpoint.
Limitation	n/a	

Scenario: Using cTry to build Try/Catch/Finally blocks for exception handling

In this scenario, the content of each file sent from the message sender to the receiver is checked and if any file does not meet the content requirement, an exception is thrown and the relevant information is displayed on the console.



Dropping and linking components

1. From the **Messaging** folder of the **Palette**, drop two**cFile** components onto the design workspace, one as the message sender and the other as the message receiver.

2. From the **Exception** folder, drop a **cTry** component onto the design workspace to build Try, Catch and Finally blocks.

3. From the **Processor** folder, drop two **cProcessor** components onto the design workspace.

4. Link the **cFile** component serving as message sender to the **cTry** component using a **Row** > **Route** connection.

5. Link the **cTry** component to one **cProcessor** using a **Row** > **Try** connection. This **cProcessor** component will throw an exception if any file coming via this connection does not contain the required content.

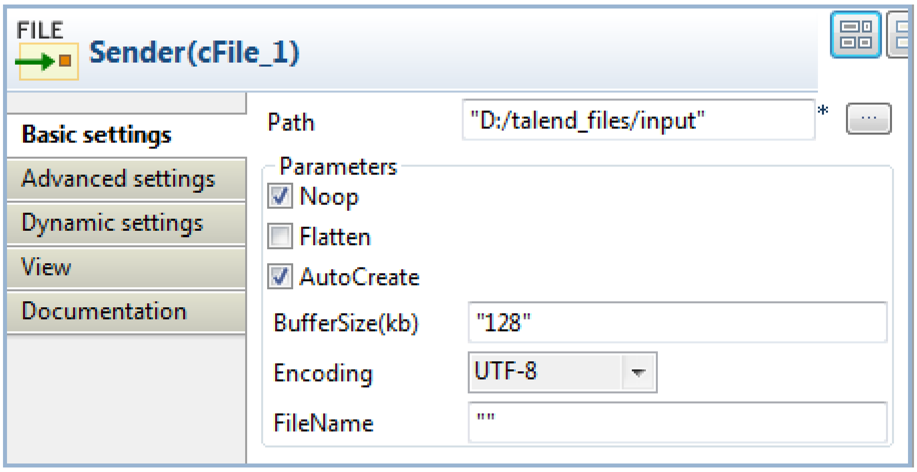
6. Link the **cTry** component to the other **cProcessor** component using a **Row** > **Catch** connection to catch the exception. This **cProcessor** component will display the information related to the exception and the file name that does not contain the required content.

7. Link the **cTry** component to the receiving **cFile** component using a **Row** > **Finally** connection.

8. Label the components according to their roles in the Route.

Configuring the components and connections

1. Double-click the **cFile** component labeled *Sender* to open its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/input"

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

Encoding UTF-8

FileName ""

2. In the **Path** field, fill in or browse to the path to the folder that holds the source files.

3. From the **Encoding** list, select the encoding type of your source files. Leave the other parameters as they are.

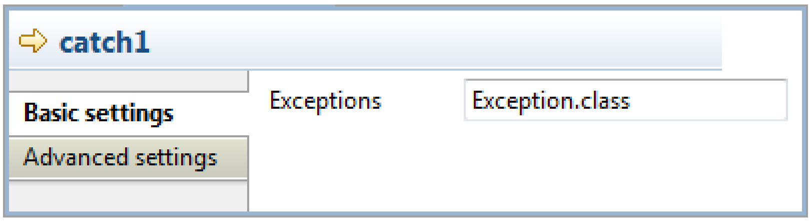
4. Repeat these step to define the output file path and encoding type in the **Basic settings** view of the other **cFile** component, which is labeled *Receiver*.

5. Double-click the **cProcessor** component labeled *Throw_exception* to open its **Basic settings** view in the **Component** tab, and customize the code in the **Code** area to throw an exception and display relevant information if any file coming via the **try** connection does not meet the content requirement, as follows:

```
String body = exchange.getIn().getBody(String.class);
System.out.println("\nTrying: "+body);
Exception e = new Exception("Only 'Talend Integration Solutions' is
 acceptable. Please check the file:");
if(!"Talend Integration Solutions".equals(body)){
 throw e;
}else{
    System.out.println("File is good.");
}
```

6. Click the **catch** connection and then the **Component** tab to open its **Basic settings** view, and fill the **Expression** field with an expression to specify the type of exception to catch.

   In this scenario, fill in `Exception.class` to catch any exception thrown.



Exceptions Exception.class

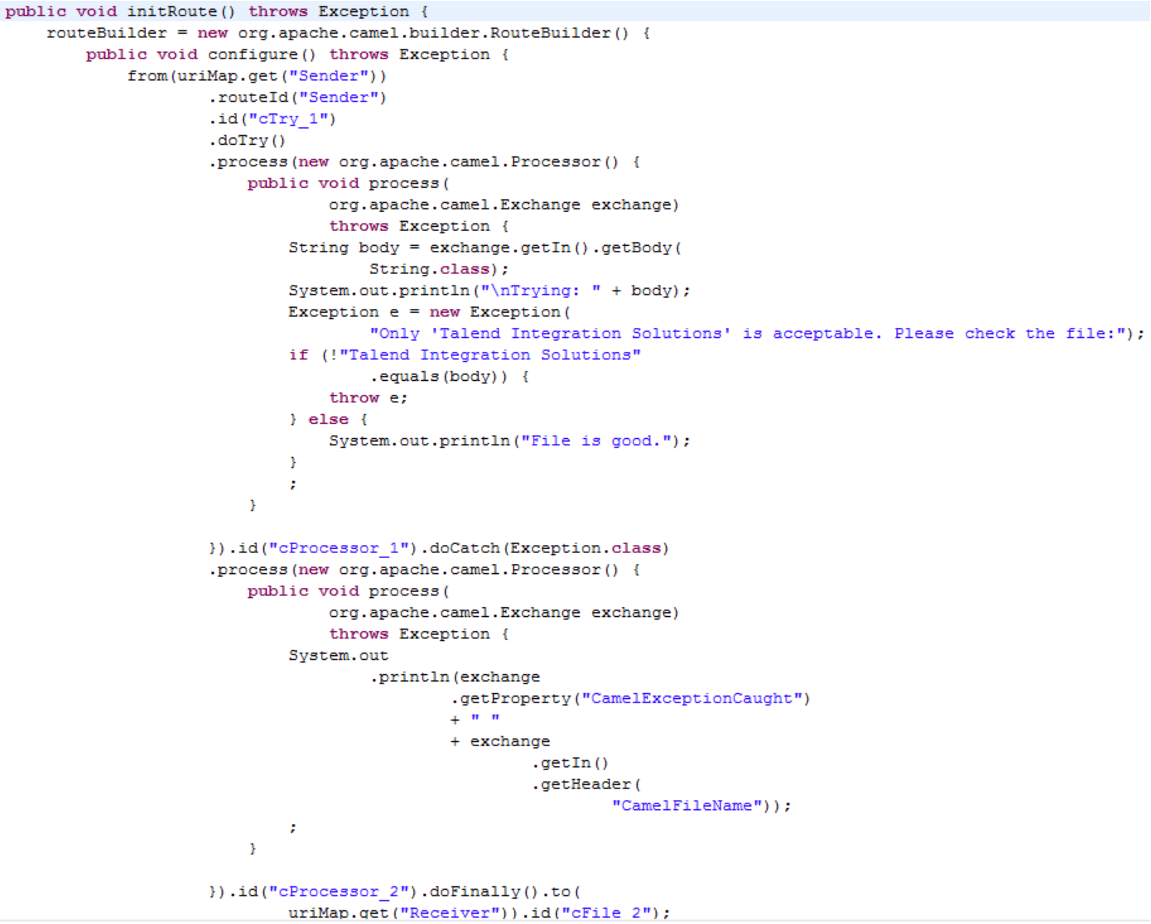
7. Double-click **cProcessor** component labeled *Show_exception* to open its **Basic settings** view in the **Component** tab, and customize the code in the **Code** area to display the exception information and the related file name, as follows:

```
System.out.println(exchange.getProperty("CamelExceptionCaught")+
" " + exchange.getIn().getHeader("CamelFileName"));
```

8. Click **Ctrl+S** to save your Route.

Viewing code and executing the Route

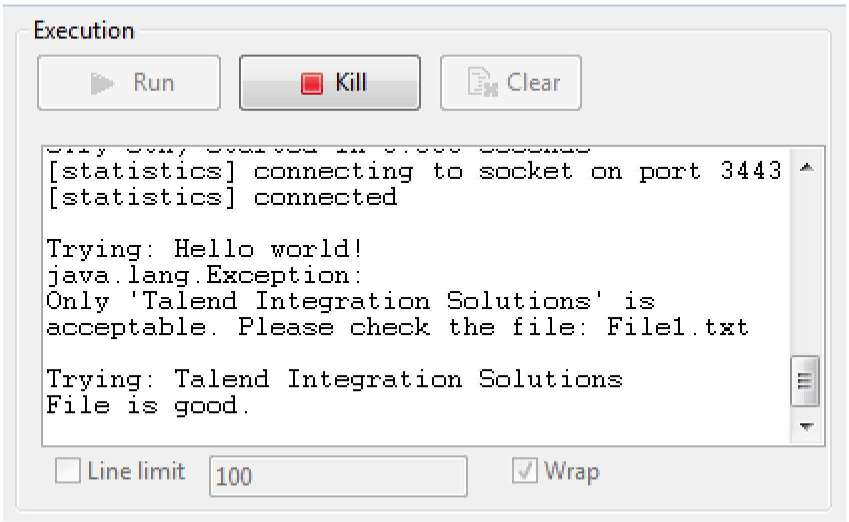
1. Click the **Code** tab at the bottom of the design workspace to check the generated code.



```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender"))
                    .routeId("Sender")
                    .id("cTry_1")
                    .doTry()
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            String body = exchange.getIn().getBody(
                                    String.class);
                            System.out.println("\nTrying: " + body);
                            Exception e = new Exception(
                                    "Only 'Talend Integration Solutions' is acceptable. Please check the file:");
                            if (!"Talend Integration Solutions"
                                    .equals(body)) {
                                throw e;
                            } else {
                                System.out.println("File is good.");
                            }
                            ;
                        }

                    }).id("cProcessor_1").doCatch(Exception.class)
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out
                                    .println(exchange
                                            .getProperty("CamelExceptionCaught")
                                            + " "
                                            + exchange
                                                    .getIn()
                                                    .getHeader(
                                                            "CamelFileName"));
                            ;
                        }

                    }).id("cProcessor_2").doFinally().to(
                            uriMap.get("Receiver")).id("cFile_2");
```

As shown above, while messages are routed `from` the sender `.to` the receiver, `.doTry()`, `.doCatch()` and `.doFinally()` blocks are built by `cTry_1`. Thus, when any file does not meet the content requirement, an exception is thrown and caught, before each file is finally routed to the receiver.

2. Press **F6** to execute the Route.



```
[statistics] connecting to socket on port 3443
[statistics] connected

Trying: Hello world!
java.lang.Exception:
Only 'Talend Integration Solutions' is
acceptable. Please check the file: File1.txt

Trying: Talend Integration Solutions
File is good.
```

RESULT: When a file that does not meet the content requirement is detected, an exception is thrown, and the exception information is displayed on the console. Regardless of the exception, all the files from the sender are sent to the receiver.

Messaging components

This chapter details the major components that you can find in the **Messaging** family from the **Palette** of the **Mediation** perspective of *Talend Open Studio for ESB*.

The **Messaging** family groups components that provide access to messaging endpoints, file systems, repository of code, and so on.

cBean

cBean properties

Component Family	Transformation	
Function	**cBean** invokes a Java beans that is stored in the **Code** node of the **Repository**.	
Purpose	**cBean** allows you to invoke a beans that is stored in the **Code** node of the **Repository**.	
Basic settings	*Bean class*	Enter the name of a bean class that is stored in the **Code** node of the **Repository**. For more information about creating and using Java Beans, see *Talend Open Studio for ESB* **User Guide**.
	Specify the method	Select this check box to enter the name of a method to be included in the bean.
Usage	**cBean** allows you to invoke a beans that is stored in the **Code** node of the **Repository**.	
Limitation		

Related Scenario

For a related scenario, see:

- **cConvertBodyTo**: the section called “Scenario: Converting the body of an XML file into an org.w3c.dom.Document.class”.

cCXF

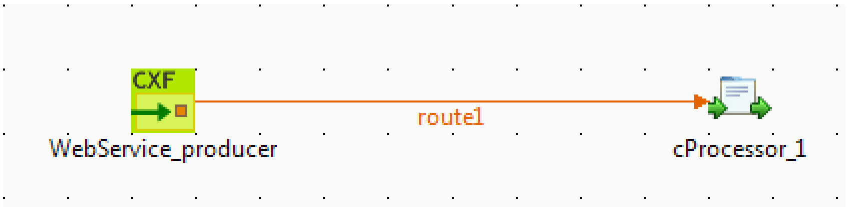
cCXF properties

Component Family	Messaging	
Function	**cCXF** provides integration with Apache CXF for connecting to JAX-WS services.	
Purpose	**cCXF** is used to provide or consume a Web service from a WSDL file or a Java class.	
Basic settings	*CXF Configuration/ Address*	The service endpoint URL where the Web service is provided.
	CXF Configuration/ Type	Select which type you want to use to provide Web service. Either **wsdlURL** or **serviceClass**. **wsdlURL**: Select this type to provide the Web service from a WSDL file. **serviceClass**: Select this type to provide the Web service from an SEI (Service Endpoint Interface) Java class.
	CXF Configuration/ WSDL File	This field displays when the **wsdlURL** service type is selected. Browse to or enter the path to the WSDL file to be used to provide the Web service.
	CXF Configuration/ Service Class	This field displays when the **serviceClass** service type is selected. Enter the name of the service class to be used to provide the Web service.
	CXF Configuration/ Dataformat	The exchange data style. **MESSAGE**, **PAYLOAD**, or **POJO**. **MESSAGE** is the raw message that is received from the transport layer. **PAYLOAD** is the message payload, the contents of the `soap:body`. **POJO**s (Plain Old Java Objects) are the Java parameters to the method being invoked on the target server.
	Service	Select this check box to specify the service port. This option is useful especially when there are multi service ports in the WSDL or service class.
	Service Name	The service name this service is implementing. It maps to the `wsdl:service@name` in the format of `ns:SERVICE_NAME` where `ns` is a namespace prefix valid at this scope.
	Port Name	The endpoint name this service is implementing. It maps to the `wsdl:port@name`, in the format of `ns:PORT_NAME` where `ns` is a namespace prefix valid at this scope.

	ESB Features/Use Service Locator	Maintains the availability of the service to help meet demands and service level agreements (SLAs). The **Custom Properties** table appears when the **Use Service Locator** check box is selected. Click ✚ to add as many properties as needed to the table. Enter the name and the value of each property in the **Property Name** field and the **Property Value** field respectively to identify the service.For more information, see *Talend ESB Runtime Configuration Guide* for how to install and configure the Service Locator.
	ESB Features/Use Service Activity Monitor	Captures events and stores this information to facilitate in-depth analysis of service activity and track-and-trace of messages throughout a business transaction. This can be used to analyze service response times, identify traffic patterns, perform root cause analysis and more. This feature is not supported when **MESSAGE** is used as the processing mode. When **MESSAGE** is selected in the **Dataformat** field, the **Use Service Activity Monitor** check box is disabled.
Advanced settings	*Arguments*	Set the optional arguments in the corresponding table. Click **[+]** as many times as required to add arguments to the table. Then click the corresponding **Value** field and enter a value. See the site http://camel.apache.org/cxf.html for available URI options.
Usage	**cCXF** can be a start, middle or end component in a Route.	
Limitation	Multiple **cCXF** components with the same label in a Route is not supported.	

Scenario 1: Providing a Web service using cCXF from a WSDL file

In this scenario, a Web service is produced by a **cCXF** component using a WSDL file.



Dropping and linking the components

This use case requires one **cCXF** component and one **cProcessor** component.

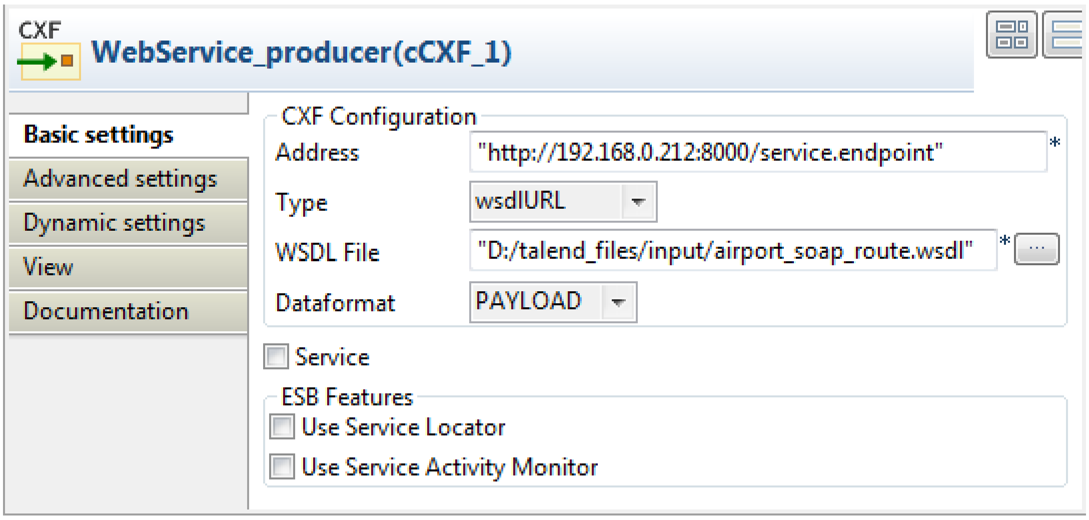
1. From the **Palette**, expand the **Messaging** folder, and drop a **cCXF** component onto the design workspace.
2. Expand the **Processor** folder, and drop a **cProcessor** component onto the design workspace.
3. Right-click the **cCXF** component, select **Row** > **Route** from the contextual menu and click the **cProcessor** component.

4. Label the **cCXF** component for better identification of its functionality.

Configuring the components

In this scenario, the **cProcessor** component is used only to enable the **cCXF** component to function as a service producer. Therefore, it does not need any configuration.

1. Double-click the **cCXF** component to display its **Basic settings** view in the **Component** tab.



CXF WebService_producer(cCXF_1)

CXF Configuration

Address "http://192.168.0.212:8000/service.endpoint" *

Type wsdlURL

WSDL File "D:/talend_files/input/airport_soap_route.wsdl" *

Dataformat PAYLOAD

□ Service

ESB Features

□ Use Service Locator

□ Use Service Activity Monitor

2. In the **Address** field, type in the service endpoint URL for the Web service to be provided, *http://192.168.0.212:8000/service.endpoint* in this example.

3. From the **Type** list, select **wsdlURL** to enable producing the Web service from a WSDL file.

4. In the **Wsdl File** field, browse to or type in the path to the WSDL file to be used.

5. From the **Dataformat** list, select **PAYLOAD** mode for the **wsdlURL** data format.

6. Press **Ctrl+S** to save your route.

Viewing code and executing the Route

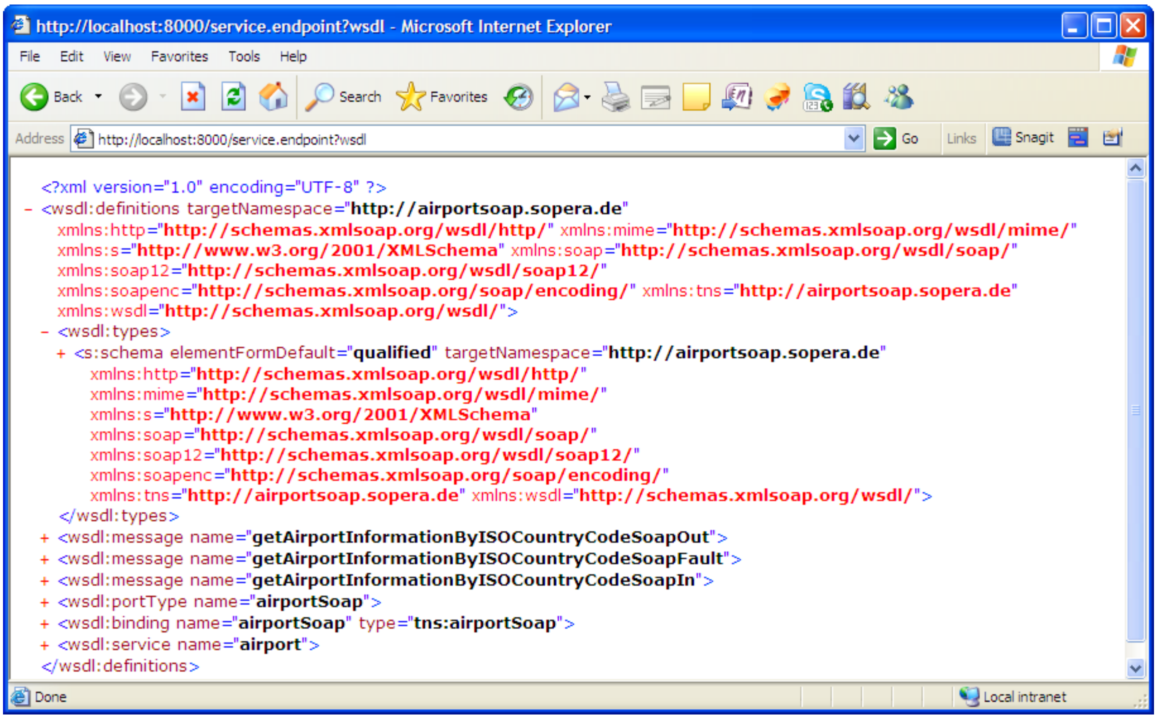
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

```
protected void initUriMap() {
    uriMap = new java.util.HashMap<String, String>();
    uriMap.put("WebService_producer", "cxf://"
            + "http://192.168.0.212:8000/service.endpoint" + "?wsdlURL="
            + "D:/talend_files/input/airport_soap_route.wsdl"
            + "&dataFormat=PAYLOAD");
```

   As shown in the code, the **cCXF** component labelled `WebService_producer` produces the Web service from an input file `airport_soap_route.wsdl` using the endpoint URL `http://192.168.0.212:8000/service.endpoint`.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route. You can also press **F6** to execute it.

RESULT: The service is successfully started. You can access it from a Web browser using the service endpoint URL followed by `?wsdl`.



http://localhost:8000/service.endpoint?wsdl - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8000/service.endpoint?wsdl
Go Links Snagit
Done
Local intranet

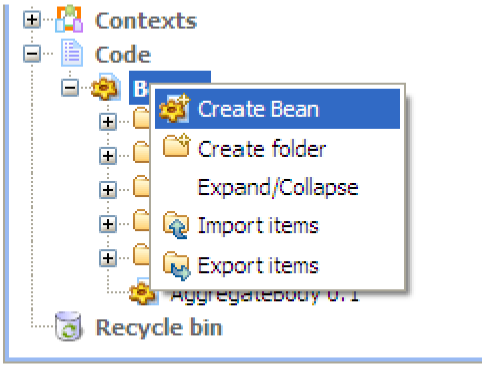
```
  <?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="http://airportsoap.sopera.de"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tns="http://airportsoap.sopera.de"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  - <wsdl:types>
    + <s:schema elementFormDefault="qualified" targetNamespace="http://airportsoap.sopera.de"
        xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
        xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
        xmlns:s="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:tns="http://airportsoap.sopera.de" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    </wsdl:types>
  + <wsdl:message name="getAirportInformationByISOCountryCodeSoapOut">
  + <wsdl:message name="getAirportInformationByISOCountryCodeSoapFault">
  + <wsdl:message name="getAirportInformationByISOCountryCodeSoapIn">
  + <wsdl:portType name="airportSoap">
  + <wsdl:binding name="airportSoap" type="tns:airportSoap">
  + <wsdl:service name="airport">
  </wsdl:definitions>
```

Scenario 2: Providing a Web service using cCXF from a Java class

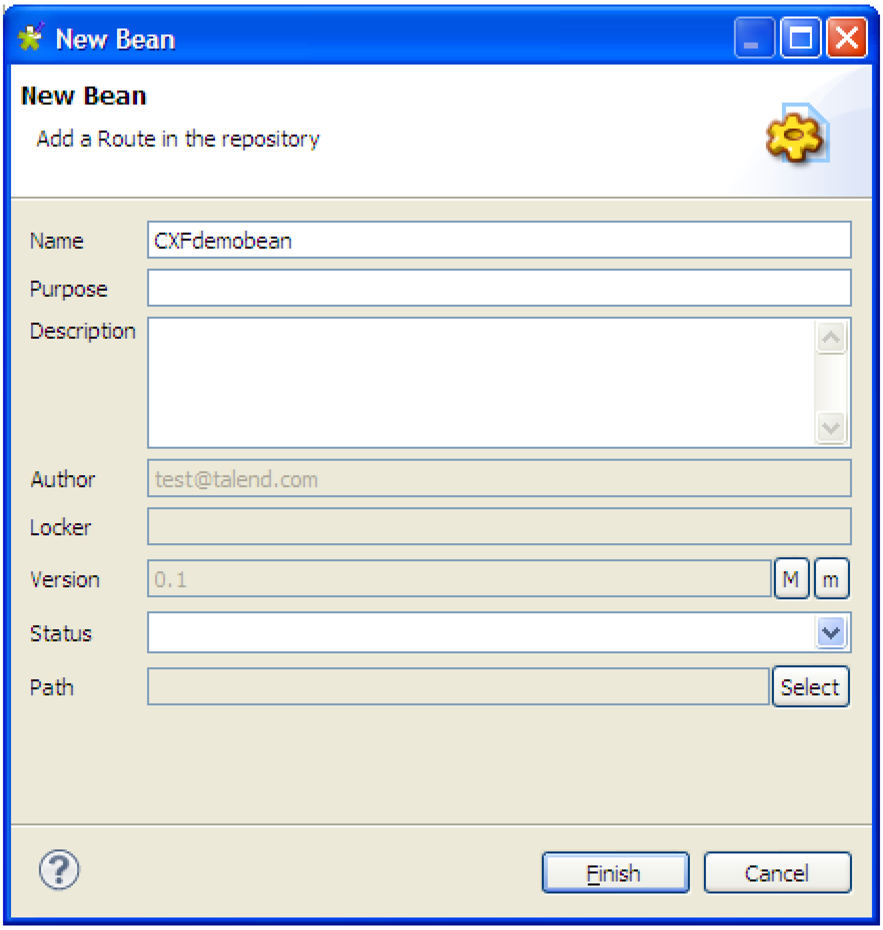
In this scenario, a Web service is provided from a Java class file using a **cCXF** component.

Creating a Java class

1. From the repository tree view, expand the **Code** node and right click the **Beans** node. In the contextual menu, select **Create Bean**.



2. The **New Bean** wizard opens. In the **Name** field, type in a name for the bean, for example, *CXFdemobean*. Click **Finish** to close the wizard.



New Bean

New Bean

Add a Route in the repository

Name: CXFdemobean

Purpose:

Description:

Author: test@talend.com

Locker:

Version: 0.1

Status:

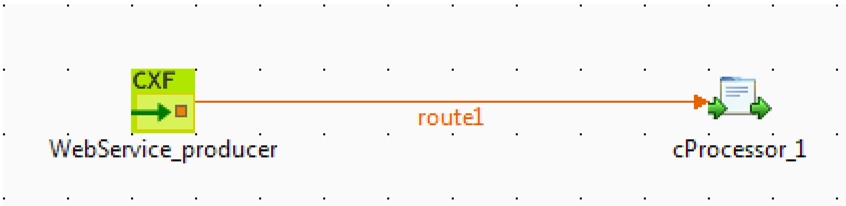
Path:

3. Change the class type to `interface`, change the return type to `string` and remove the message body.

```
package beans;

public interface CXFdemobean {
    public String helloExample(String message) ;
}
```

4. Press **Ctrl+S** to save your bean.

Dropping and linking the components



This use case requires one **cCXF** component and one **cProcessor** component.

1. From the **Palette**, expand the **Messaging** folder, select the **cCXF** component and drop it onto the design workspace.

2. Expand the **Processor** folder, select the **cProcessor** component and drop it onto the design workspace.

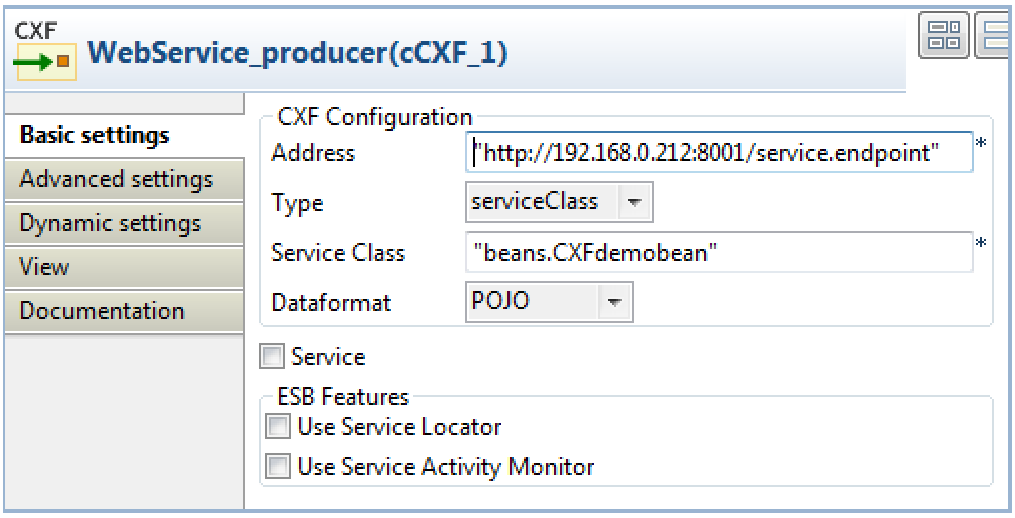
3. Right-click the **cCXF** component, select **Row** > **Route** in the contextual menu and click the **cProcessor** component.

4. Label the components for better identification of their functionality.

Configuring the components

In this scenario, the **cProcessor** component is used only to enable the **cCXF** component to function as a service producer. Therefore, it does not need any configuration.

1. Double-click the **cCXF** component to display its **Basic settings** view in the **Component** tab.



CXF Configuration

Address "http://192.168.0.212:8001/service.endpoint" *

Type serviceClass

Service Class "beans.CXFdemobean" *

Dataformat POJO

□ Service

ESB Features

□ Use Service Locator

□ Use Service Activity Monitor

2. In the **Address** field, type in the service endpoint URL for the Web service to be provided, *http://192.168.0.212:8001/service.endpoint* in this example.

3. From the **Type** from, select **serviceClass** to start the Web service from a Java class.

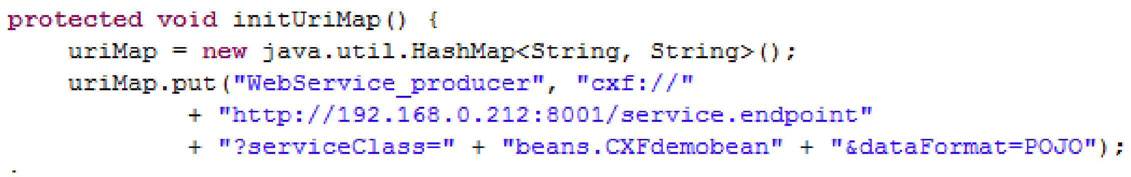
4. In the **Service Class** field, specify the predefined bean class, *CXFdemobean* in this example.

5. From the **Dataformat** list, select **POJO** as the **serviceClass** service data format.

6. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

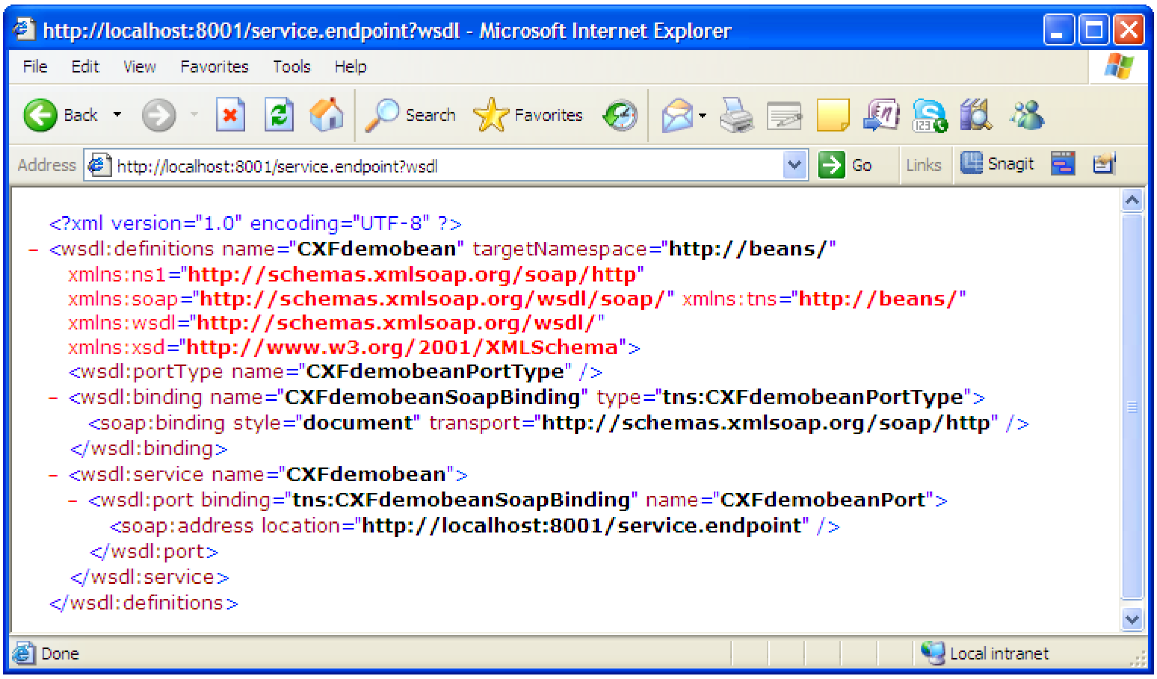


```
protected void initUriMap() {
    uriMap = new java.util.HashMap<String, String>();
    uriMap.put("WebService_producer", "cxf://"
            + "http://192.168.0.212:8001/service.endpoint"
            + "?serviceClass=" + "beans.CXFdemobean" + "&dataFormat=POJO");
```

As shown in the code, the **cCXF** component labelled `WebService_producer` produces the Web service from an predefined bean `beans.CXFdemobean` using the endpoint URL `http://192.168.0.212:8001/service.endpoint`.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route. You can also press **F6** to execute it.

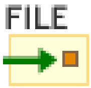
   RESULT: The service is successfully started. You can access it from a Web browser using the service endpoint URL followed by `?wsdl`.



http://localhost:8001/service.endpoint?wsdl - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address http://localhost:8001/service.endpoint?wsdl
Go
Links
Snagit
Done
Local intranet

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions name="CXFdemobean" targetNamespace="http://beans/"
    xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://beans/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:portType name="CXFdemobeanPortType" />
  - <wsdl:binding name="CXFdemobeanSoapBinding" type="tns:CXFdemobeanPortType">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
    </wsdl:binding>
  - <wsdl:service name="CXFdemobean">
    - <wsdl:port binding="tns:CXFdemobeanSoapBinding" name="CXFdemobeanPort">
        <soap:address location="http://localhost:8001/service.endpoint" />
      </wsdl:port>
    </wsdl:service>
  </wsdl:definitions>
```

cFile

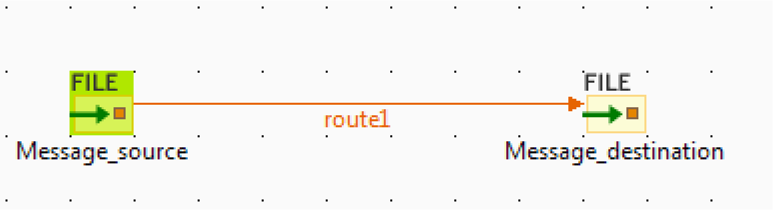


cFile properties

Component Family	Messaging	
Function	**cFile** provides access to file systems.	
Purpose	**cFile** allows files to be processed by any other Camel components or messages from other components to be saved to disk.	
Basic settings	*Path*	Path to the file or files to be accessed or saved.
	Parameters/Noop	Select this check box to keep the file or files in the original folder after being read.
	Parameters/Flatten	Select this check box to flatten the file name path to strip any leading paths. This allows you to consume recursively into sub-directories, but when you, for example, write the files to another directory, they will be written in a single directory.
	Parameters/AutoCreate	Select this check box to create the directory specified in the **Path** field automatically if it does not exist.
	Parameters/ BufferSize(kb)	Write buffer sized in bytes.
	Encoding	Specify the encoding of the file, **ISO-8859-15**, **UTF-8**, or **CUSTOM**.
	FileName	The name of the file to be processed. Use this option if you want to consume only a single file in the specified directory.
Advanced settings	*Advanced*	Set the optional arguments in the corresponding table. Click **[+]** as many times as required to add arguments to the table. Then click the corresponding **Value** field and enter a value. See the site http://camel.apache.org/file2.html for available URI options.
Usage	**cFile** can be a start, middle or end component in a Route.	
Limitation	n/a	

Scenario: Reading files from one directory and writing them to another

In this scenario, an input **cFile** component is configured to visit a local file directory and send the files in the directory to an output **cFile** component which writes the files in another directory.

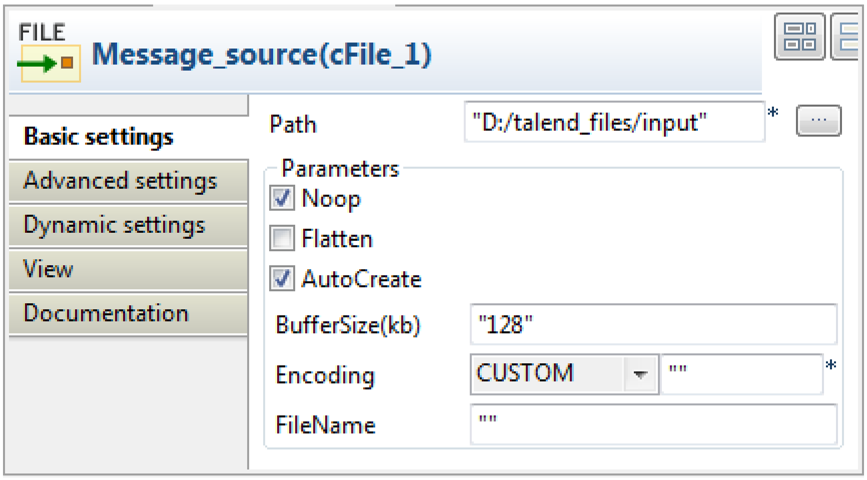


Dropping and linking the components

1. From the **Palette**, expand the **Messaging** folder and select the **cFile** component. Drop one as the input component and another as the output component onto the design workspace.
2. Right-click the input **cFile** component, select **Row** > **Route** in the contextual menu and click the output **cFile** component.
3. Label the components to better identify their respective functionality.

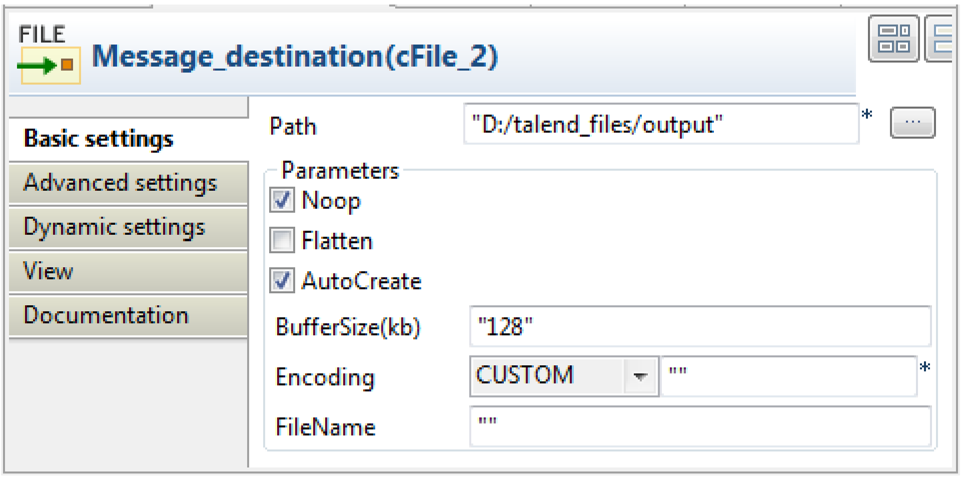
Configuring the components

1. Double-click the input **cFile** component to display its **Basic settings** view in the Component tab.



Path "D:/talend_files/input" *
Parameters
☑ Noop
☐ Flatten
☑ AutoCreate
BufferSize(kb) "128"
Encoding CUSTOM "" *
FileName ""

2. In the **Path** field, browse to or enter the input file path, and leave the other parameters as they are.
3. Double-click the output **cFile** component to display its **Basic settings** view in the **Component** tab.



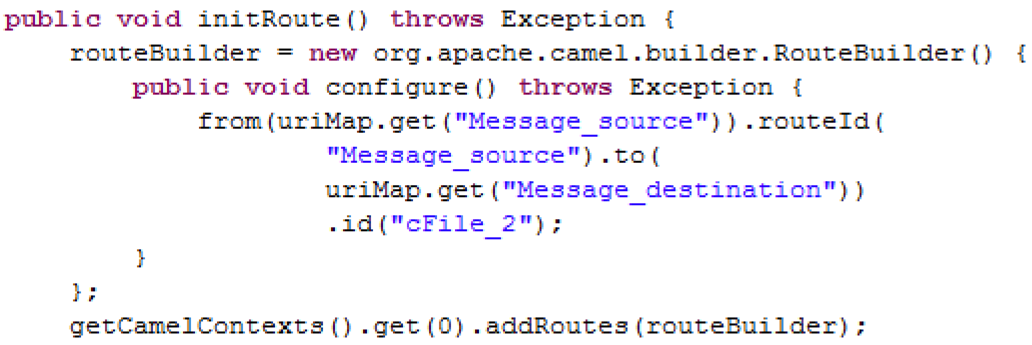
Path "D:/talend_files/output" *
Parameters
☑ Noop
☐ Flatten
☑ AutoCreate
BufferSize(kb) "128"
Encoding CUSTOM "" *
FileName ""

4. In the **Path** field, browse to or enter the output file path, as shown above. Leave the other parameters as they are.

5. Press **Ctrl+S** to save your route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.



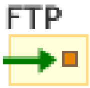
```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Message_source")).routeId(
                    "Message_source").to(
                    uriMap.get("Message_destination"))
                    .id("cFile_2");
        }
    };
    getCamelContexts().get(0).addRoutes(routeBuilder);
```

As shown in the code, a message route is built `from` one endpoint `.to` another.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route. You can also press **F6** to execute it.

RESULT: The input files are written to specified output directory.

cFtp



cFtp properties

Component Family	Messaging	
Function	**cFtp** provides access to remote file systems over the FTP, FTPS and SFTP protocols.	
Purpose	**cFtp** allows data exchange over remote file systems.	
Basic settings	*Parameters/type*	Select the file transfer protocol, **ftp** or **sftp**, **ftps**.
	Parameters/server	Type in the remote server address to be accessed.
	Parameters/port	Type in the port number to be accessed.
	Parameters/username	Type in the user authentication information.
	Parameters/password	Type in the user authentication information.
	Parameters/directory	Enter the directory you want to access on the remote server. If not specified, the root directory will be accessed.
Advanced settings	*Advanced*	Set the optional arguments in the corresponding table. Click **[+]** as many times as required to add arguments to the table. Then click the corresponding **Value** field and enter a value. See the site http://camel.apache.org/ftp.html for available URI options.
Usage	**cFtp** can be a start, middle or end component in a Route.	
Limitation	n/a	

Related scenario:

No scenario is available for this component yet.

cHttp

cHttp properties

Component Family	Messaging	
Function	**cHttp** provides Http-based endpoints for consuming external Http resources, i.e. as a client to call external servers using Http.	
Purpose	**cHttp** is designed to build a client endpoint to call external Http resources using Http.	
Basic settings	*Uri*	The URI of the Http resource to call.
	Method	List of the Http request methods.
	Get	Retrieve the information identified by the request URI: **Parameters**: click the **[+]** button to add lines as needed and define the key and value in the table. **Encoder Charset**: enter the encoder charset in the field.
	Post	Request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the request URI: **Plain text**: type in the text in the **Content** box as the request message. **Form Style**: click the **[+]** button to add lines as needed and define the key and value in the **Parameters** table. Also, enter the encoder charset in the **Encoder Charset** field. **Use Message Body**: use the incoming message body as the Http request.
	Put	Request that the enclosed entity be stored under the supplied request URI.
	Delete	Request that the origin server delete the resource identified by the request URI.
	Head	Identical to GET except that the server MUST NOT return a message body in the response: **Parameters**: click the **[+]** button to add lines as needed and define the key and value in the table. **Encoder Charset**: enter the encoder charset in the field.
	Options	Represent a request for information about the communication options available on the request/response chain identified by the request URI.
	Trace	Invoke a remote, application-layer loop-back of the request message.

Advanced settings	*Headers*	Click the **[+]** button to add lines as needed and define the key and value for headers.
Usage	**cHttp** provides Http based endpoints for consuming external Http resources, i.e. as a client to call external servers using Http.	
Limitation		

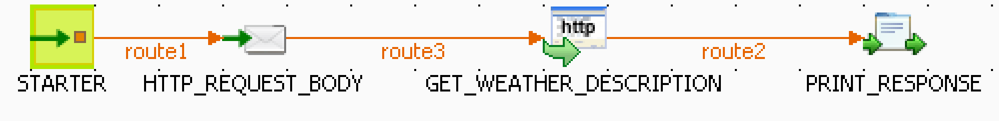
Scenario: Retrieving the content of a remote file

In this scenario, **cHttp** is used to request the body of a weather condition definition file that is available at http://wsf.cdyne.com/WeatherWS/Weather.asmx.

Dropping and linking the components

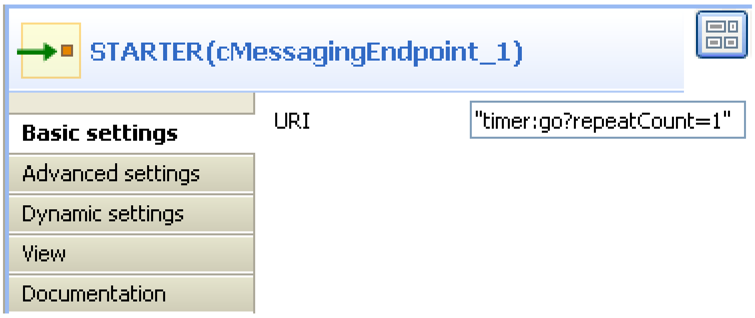
1. Drop the following components from the **Palette** onto the workspace: **cMessagingEndpoint**, **cSetBody**, **cHttp** and **cProcessor**, labelled as **STARTER**, **HTTP_REQUEST_BODY**, **GET_WEATHER_DESCRIPTION** and **PRINT_RESPONSE** respectively.

2. Link the components using a **Row** > **Route** connection.



Configuring the components

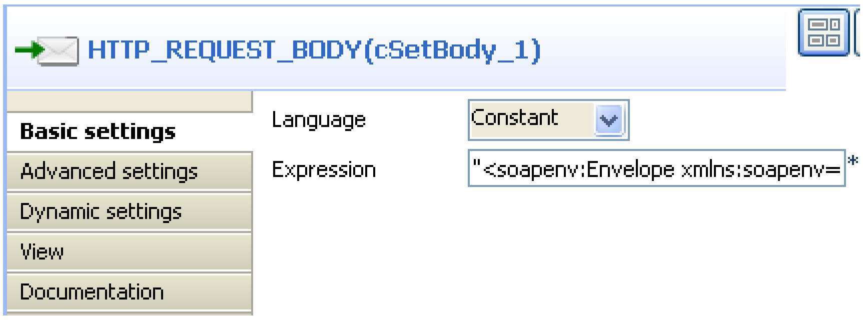
1. Double-click **cMessagingEndpoint** to open its **Basic settings** view in the **Component** tab.


STARTER(cMessagingEndpoint_1)

URI "timer:go?repeatCount=1"

2. In the **URI** field, enter `timer:go?repeatCount=1` to define a timer for starting message exchanges. In this example, only one message exchange will be carried out due to the setting of `repeatCount=1`.

3. Double-click **cSetBody** to open its **Basic settings** view in the **Component** tab.


HTTP_REQUEST_BODY(cSetBody_1)

Language Constant

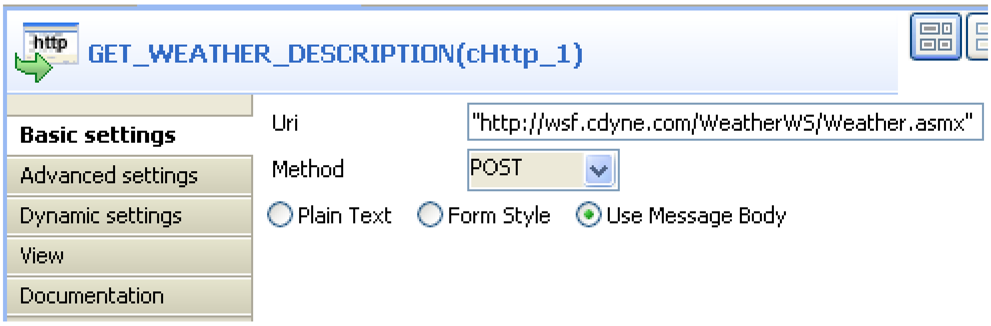
Expression "<soapenv:Envelope xmlns:soapenv=

4. In the **Language** field, select *Constant*.

5. In the **Expression** field, enter the following as the body of the request message:

```
<soapenv:Envelope xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/
envelope/\" xmlns:weat=\"http://ws.cdyne.com
/WeatherWS/\"><soapenv:Header/
><soapenv:Body><weat:GetWeatherDefinitionInformation/></soapenv:Body></
soapenv:Envelope>
```

6. Double-click **cHttp** to open its **Basic settings** view in the **Component** tab.



Uri "http://wsf.cdyne.com/WeatherWS/Weather.asmx"

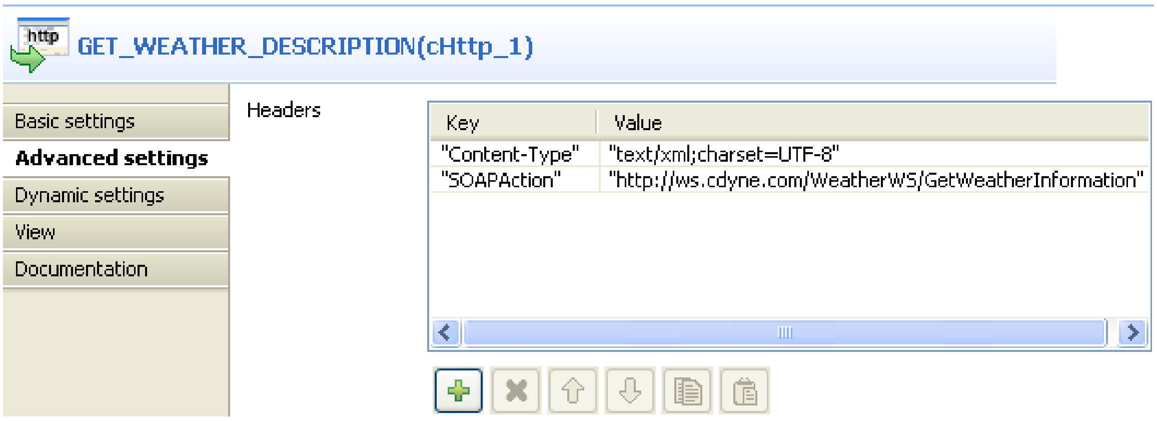
Method POST

○ Plain Text ○ Form Style ● Use Message Body

7. In the **Uri** field, enter the location of the file to fetch, *http://wsf.cdyne.com/WeatherWS/Weather.asmx* in this example.

8. Select *POST* in the **Method** list and then the **Use Message Body** box.

9. Click **Advanced settings** for further setup.



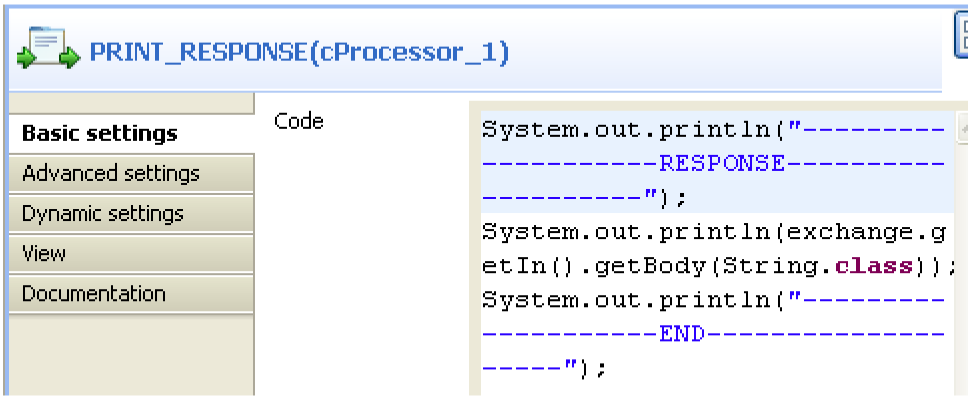
Headers

Key	Value
"Content-Type"	"text/xml;charset=UTF-8"
"SOAPAction"	"http://ws.cdyne.com/WeatherWS/GetWeatherInformation"

10. Click the **[+]** button to add two lines in the **Headers** table.

    Type in `Content-Type` and `text/xml;charset=UTF-8` for the **Key** and **Value** fields in the first line, and `SOAPAction` as well as `http://ws.cdyne.com/WeatherWS/GetWeatherInformation` in the second line.

11. Double-click **cProcessor** to open its **Basic settings** view in the **Component** tab.



Code

```
System.out.println("---------
------------RESPONSE----------
----------");
System.out.println(exchange.g
etIn().getBody(String.class));
System.out.println("---------
------------END---------------
-----");
```

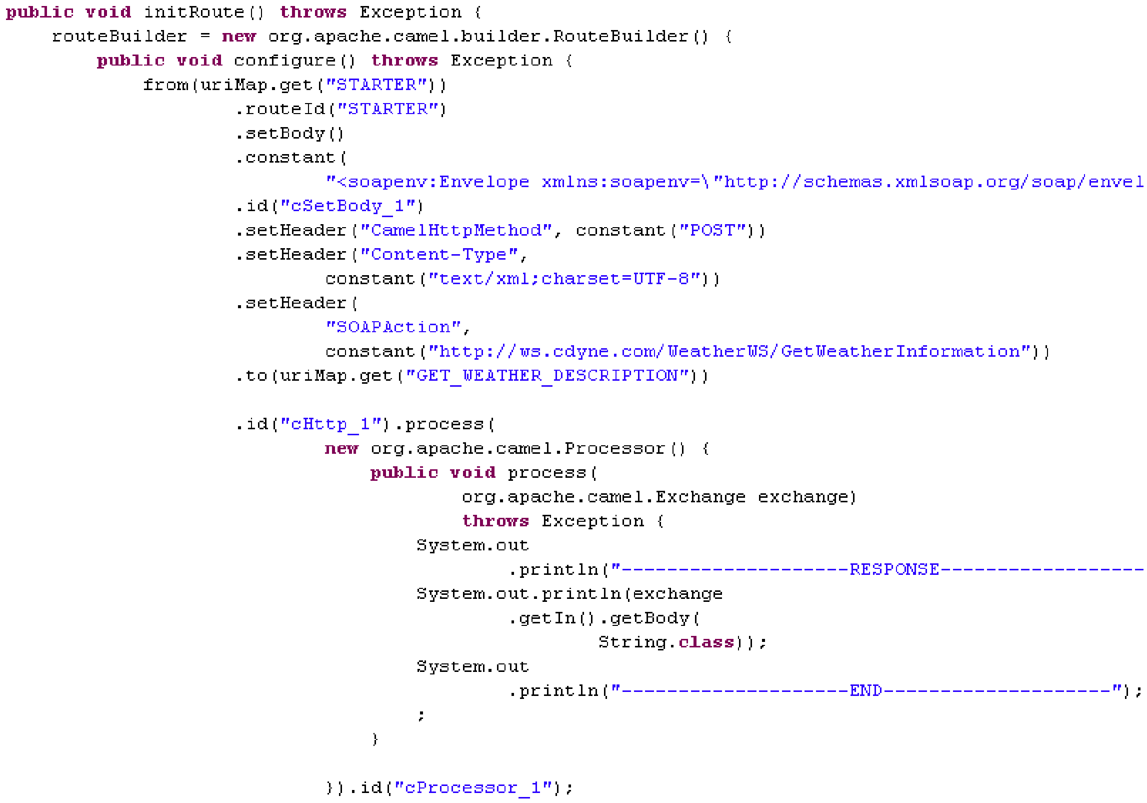
12. In the **Code** area, enter the following to print the response from the remote website, i.e. the body of the desired file:

```
System.out.println("--------------------RESPONSE--------------------");
System.out.println(exchange.getIn().getBody(String.class));
System.out.println("--------------------END---------------------");
```

13. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.

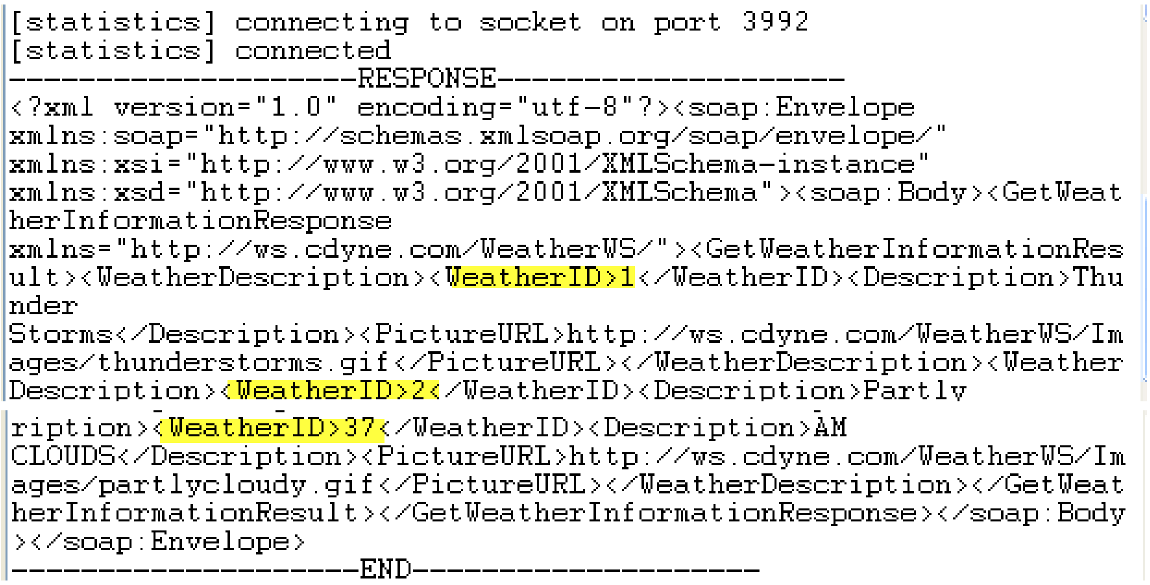


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("STARTER"))
                    .routeId("STARTER")
                    .setBody()
                    .constant(
                            "<soapenv:Envelope xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envel
                    .id("cSetBody_1")
                    .setHeader("CamelHttpMethod", constant("POST"))
                    .setHeader("Content-Type",
                            constant("text/xml;charset=UTF-8"))
                    .setHeader(
                            "SOAPAction",
                            constant("http://ws.cdyne.com/WeatherWS/GetWeatherInformation"))
                    .to(uriMap.get("GET_WEATHER_DESCRIPTION"))

                    .id("cHttp_1").process(
                            new org.apache.camel.Processor() {
                                public void process(
                                        org.apache.camel.Exchange exchange)
                                        throws Exception {
                                    System.out
                                            .println("--------------------RESPONSE------------------
                                    System.out.println(exchange
                                            .getIn().getBody(
                                                    String.class));
                                    System.out
                                            .println("--------------------END--------------------");
                                    ;
                                }

                            }).id("cProcessor_1");
```

As shown above, the message exchange starts `from` the endpoint `STARTER`, gets its body set to `<soapenv:Envelope xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\"xmlns:weat=\"http://ws.cdyne.com/WeatherWS/\"><soapenv:Header/`

`><soapenv:Body><weat:GetWeatherDefinitionInformation/></soapenv:Body></soapenv:Envelope>` at `cSetBody_1`, and then is sent out to the specified website by `cHttp_1`. Finally, the response is printed out via `cProcessor_1`.

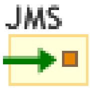
2. Press **F6** to execute the Route.



```
[statistics] connecting to socket on port 3992
[statistics] connected
--------------------RESPONSE--------------------
<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><soap:Body><GetWeat
herInformationResponse
xmlns="http://ws.cdyne.com/WeatherWS/"><GetWeatherInformationRes
ult><WeatherDescription><WeatherID>1</WeatherID><Description>Thu
nder
Storms</Description><PictureURL>http://ws.cdyne.com/WeatherWS/Im
ages/thunderstorms.gif</PictureURL></WeatherDescription><Weather
Description><WeatherID>2</WeatherID><Description>Partly
ription><WeatherID>37</WeatherID><Description>AM
CLOUDS</Description><PictureURL>http://ws.cdyne.com/WeatherWS/Im
ages/partlycloudy.gif</PictureURL></WeatherDescription></GetWeat
herInformationResult></GetWeatherInformationResponse></soap:Body
></soap:Envelope>
--------------------END--------------------
```

As shown above, the retrieved file defines up to 37 weather conditions with detailed description.

cJMS

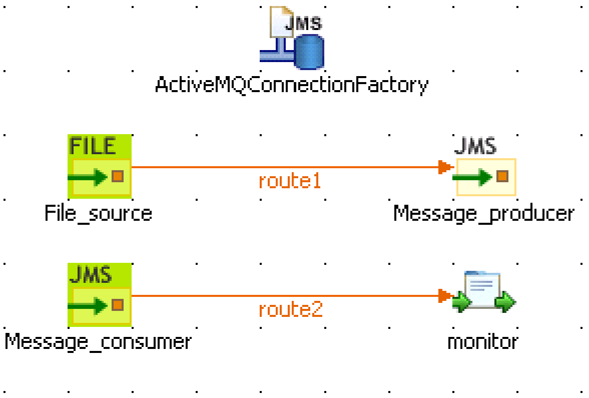


cJMS properties

Component Family	Messaging	
Function	**cJMS** allows messages to be sent to, or consumed from, a JMS Queue or Topic.	
Purpose	**cJMS** is used to send messages to, or consume messages from, a JMS Queue or Topic.	
	URI/Type	Select the messaging type, either **queue** or **topic**.
	URI/Destination	Type in a name for the JMS queue or topic.
	ConnectionFactory	Click the three-dot button and select a JMS connection factory to be used for handling messages or enter the name of the corresponding **cJMSConnectionFactory** component directly in the field.
Advanced settings	*URI Options*	Set the optional arguments in the corresponding table. Click **[+]** as many times as required to add arguments to the table. Then click the corresponding value field and enter a value. See the site http://camel.apache.org/jms.html for available URI options.
Usage	**cJMS** can be a start, middle or end component in a Route.	
Limitation	n/a	

Scenario: Sending and receiving a message from a JMS queue

In this scenario, a **cJMS** component sends messages from the local file system to a message queue in one sub-route, and the messages are then consumed by another **cJMS** component in the other sub-route.



Dropping and linking the components

1. From the **Palette**, expand the **Context** folder, and drop a **cJMSConnectionFactory** component onto the design workspace to specify the JMS connection factory for handling messages.

2. From the **Messaging** folder, drop one **cFile** and two **cJMS** components onto the design workspace.

3. From the **Processor** folder, drop a **cProcessor** component onto the design workspace.

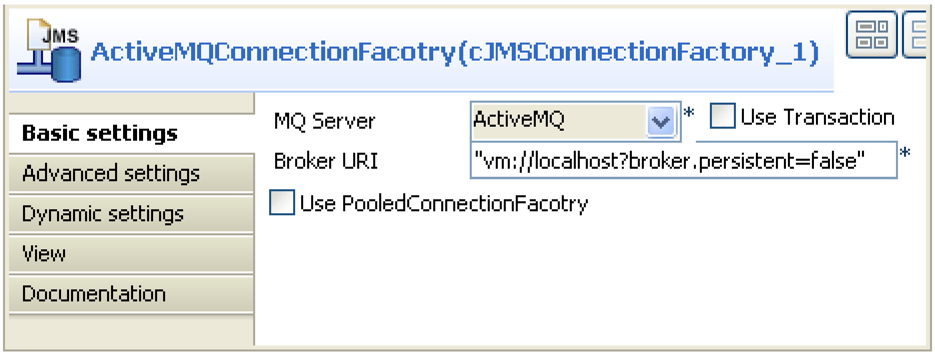
4. Connect the **cFile** component to a **cJMS** component using a **Row** > **Route** connection as the message producer sub-route.

5. Connect the other **cJMS** component to the **cProcessor** component using a **Row** > **Route** connection as the message consumer sub-route.

6. Label the components properly for better identification of their functionalities.

Configuring the components

1. Double-click the **cJMSConnectionFactory** component to display its **Basic settings** view in the **Component** tab.



ActiveMQConnectionFacotry(cJMSConnectionFactory_1)

MQ Server: ActiveMQ *

□ Use Transaction

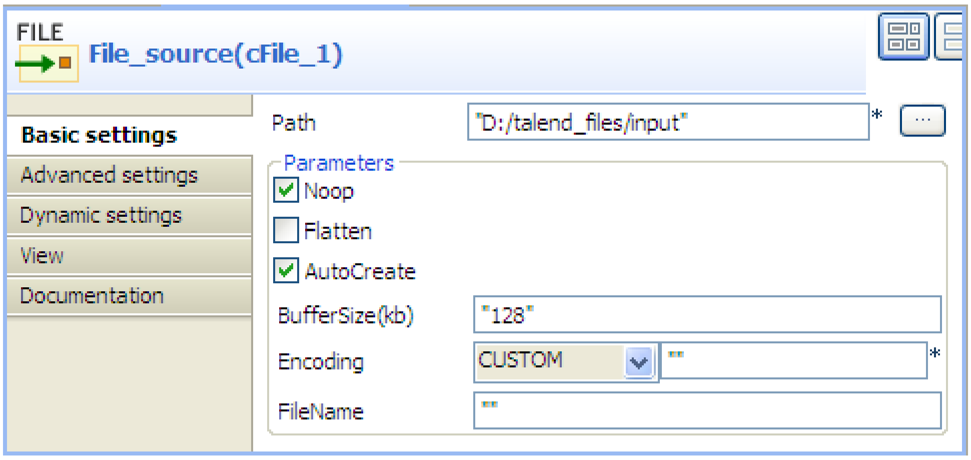
Broker URI: "vm://localhost?broker.persistent=false" *

□ Use PooledConnectionFacotry

2. From the **MQ Server** list, select an MQ server. In this use case, we use the default ActiveMQ server to handle the messages.

   In the **Broker URI** field, type in the URI of the message broker. Here we simply use the default URI *"vm://localhost?broker.persistent=false"*.

3. In the message producer sub-route, double-click the **cFile** component to display its **Basic settings** view.



File_source(cFile_1)

Path: "D:/talend_files/input" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb): "128"

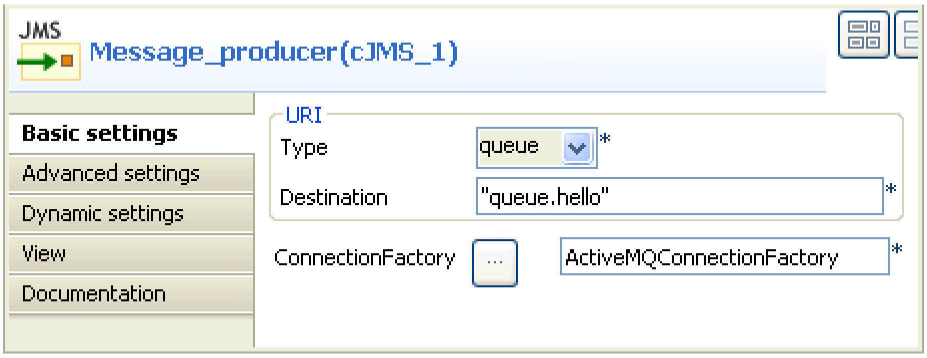
Encoding: CUSTOM "" *

FileName: ""

4. Define the properties of the **cFile** component.

   In this use case, simply specify the path to the folder that holds the source file to be sent as electronic message, and leave the other parameters as they are.

5. Double-click the **cJMS** component labeled *Message_producer* to display its **Basic settings** view.

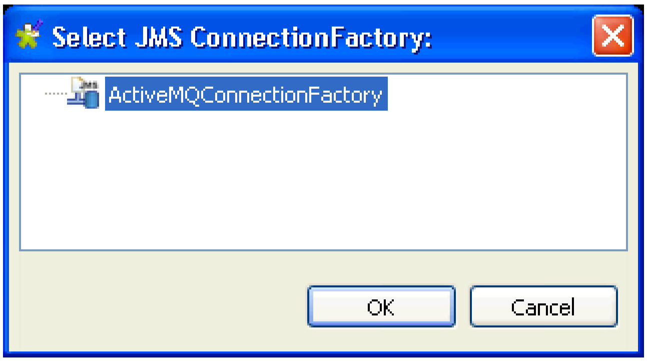


Type: queue *
Destination: "queue.hello" *
ConnectionFactory: ActiveMQConnectionFactory *

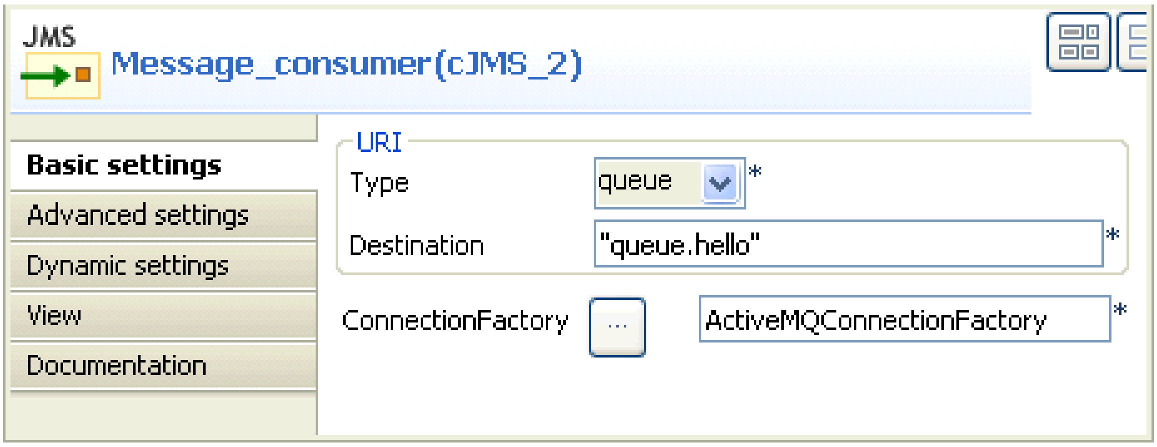
6. From the **Type** list, select **queue** to send the messages to a JMS queue.

   In the **Destination** field, type in a name for the JMS queue, `"queue.hello"` in this use case.

   Double-click the **[...]** button next to **ConnectionFactory**. Select the JMS connection factory that you have just configured in the dialog box and click **OK**. You can also enter the name of the **cJMSConnectionFactory** component directly in the field.



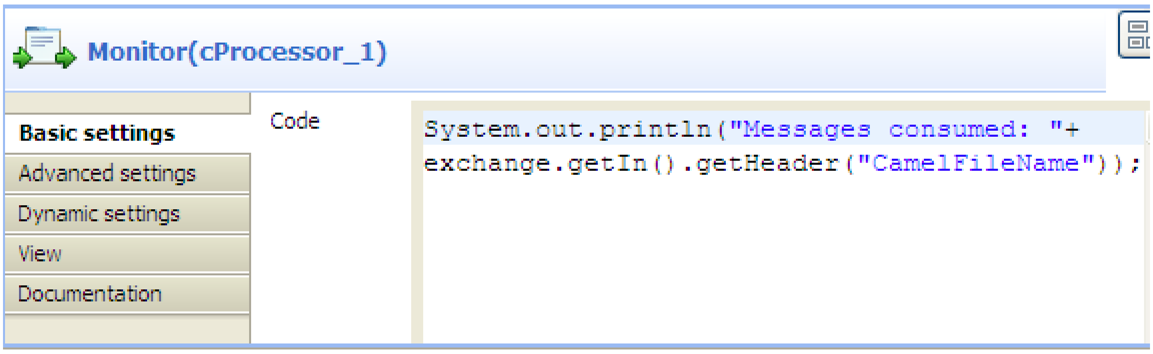
7. Switch to the message consumer sub-route, and double click the **cJMS** component labeled *Message_consumer* to display its **Basic settings** view.



Type: queue *
Destination: "queue.hello" *
ConnectionFactory: ActiveMQConnectionFactory *

8. Configure the message consumer using exactly the same parameters as in the message producer.

9. Double-click the **cProcessor** component to display its **Basic settings** view.



```
System.out.println("Messages consumed: "+
exchange.getIn().getHeader("CamelFileName"));
```

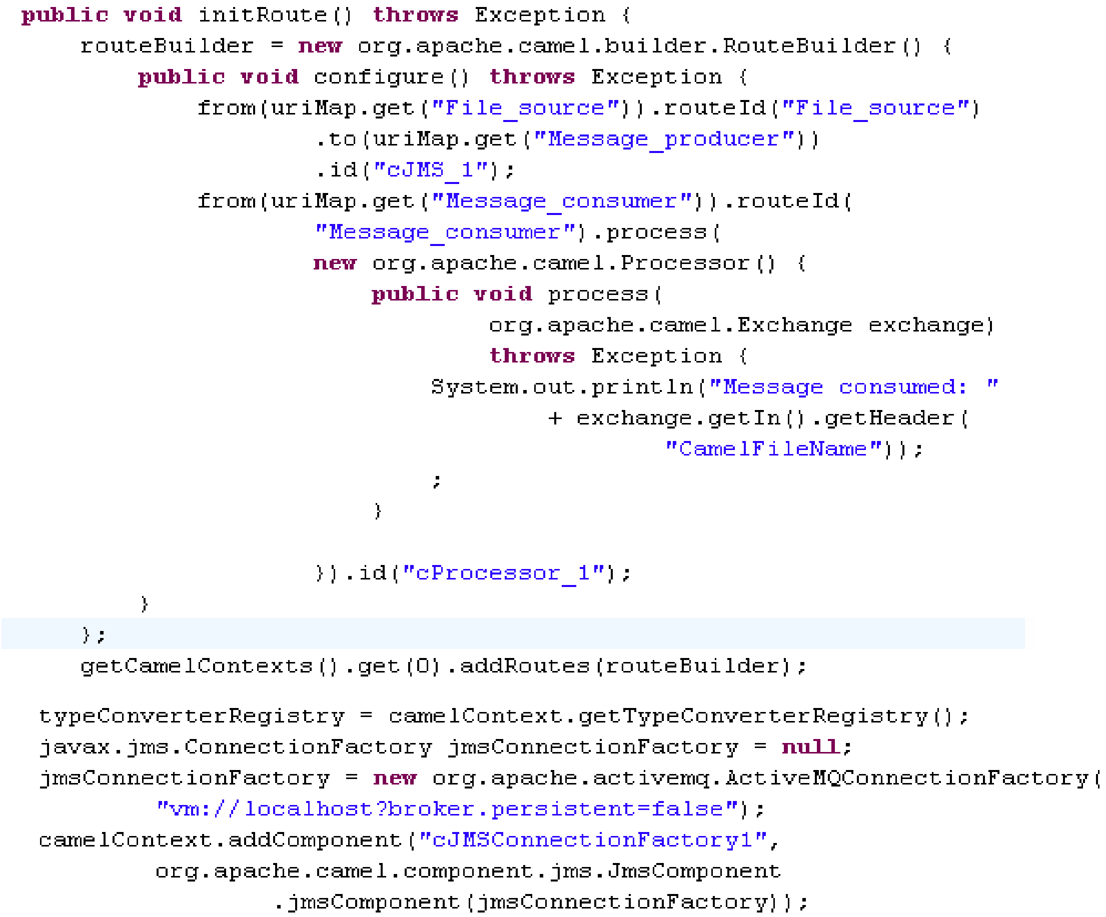
10. In the **Code** area, customize the code as shown below to display the file names of the consumed messages on the **Run** console.

```
System.out.println("Message consumed: "+
exchange.getIn().getHeader("CamelFileName"));
```

11. Press **Ctrl+S** to save your Routes.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

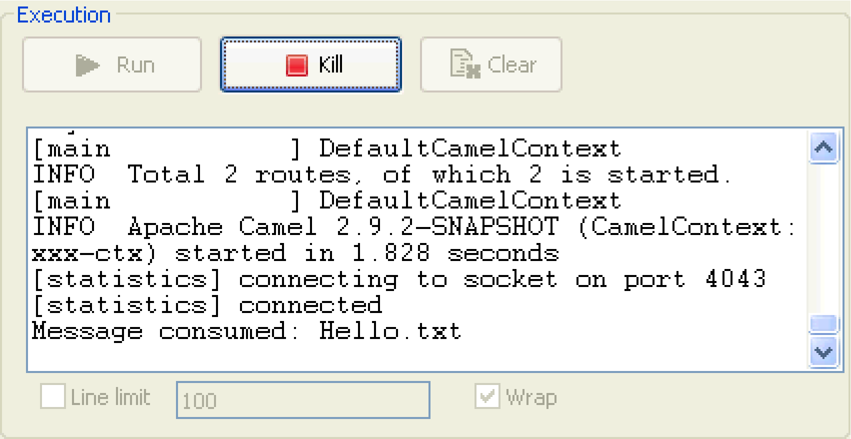


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("File_source")).routeId("File_source")
                    .to(uriMap.get("Message_producer"))
                    .id("cJMS_1");
            from(uriMap.get("Message_consumer")).routeId(
                    "Message_consumer").process(
                    new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out.println("Message consumed: "
                                    + exchange.getIn().getHeader(
                                            "CamelFileName"));
                            ;
                        }

                    }).id("cProcessor_1");
        }
    };
    getCamelContexts().get(0).addRoutes(routeBuilder);

  typeConverterRegistry = camelContext.getTypeConverterRegistry();
  javax.jms.ConnectionFactory jmsConnectionFactory = null;
  jmsConnectionFactory = new org.apache.activemq.ActiveMQConnectionFactory(
          "vm://localhost?broker.persistent=false");
  camelContext.addComponent("cJMSConnectionFactory1",
          org.apache.camel.component.jms.JmsComponent
                  .jmsComponent(jmsConnectionFactory));
```

In the partially shown code, a message route is built `from` the `File_source` `.to` the `Message_producer` which then sends the message to a message queue via a broker identified by `vm://localhost?broker.persistent=false`. The message `from` the `Message_consumer` is processed by `cProcessor_1`.

2. Click the **Run** button in the **Run** view to launch the execution of your Route. You can also press **F6** to execute it.

   RESULT: The message is received by the consumer, as shown on the **Run** console.



```
[main              ] DefaultCamelContext
INFO  Total 2 routes, of which 2 is started.
[main              ] DefaultCamelContext
INFO  Apache Camel 2.9.2-SNAPSHOT (CamelContext:
xxx-ctx) started in 1.828 seconds
[statistics] connecting to socket on port 4043
[statistics] connected
Message consumed: Hello.txt
```

Scenario: Setting up a JMS local transaction

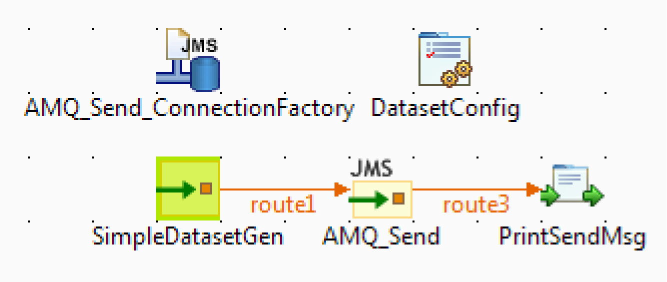
In this scenario, a local transaction with three steps is performed to send, test and consume a JMS message:

1. The first Route is used to send a *"hello world!"* message to feed the *queue.hello* JMS queue.

2. The second Route is used to test the received JMS message. This message is redelivered six times to the *queue.hello* queue and is then moved to the *Dead Letter* JMS queue. The Route is programmed to throw an exception every time an exchange is processed by the Route.

3. The last Route is used to consume the *"hello world!"* message from the *Dead Letter* JMS queue.

Sending a message to the *queue.hello* JMS queue



Procedure 1. Dropping and linking the components

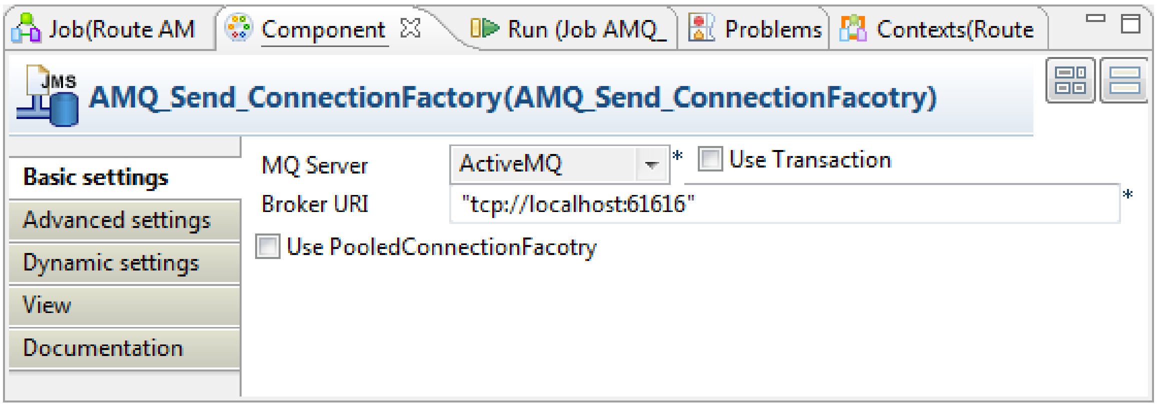
1. From the **Palette**, drop the five following components onto the design workspace: one **cJMSConnectionFactory**, one **cConfig**, one **cMessagingEndpoint**, one **cJMS** and one **cProcessor** component.

2. Connect the **cMessagingEndpoint** component to the **cJMS** using a **Row** > **Route** connection.

3. Connect the **cJMS** component to the **cProcessor** component using a **Row** > **Route** connection.

Procedure 2. Configuring the components

1. Double-click the **cJMSConnectionFactory** component labelled *AMQ_Send_ConnectionFactory* to display its **Basic settings** view in the **Component** tab.



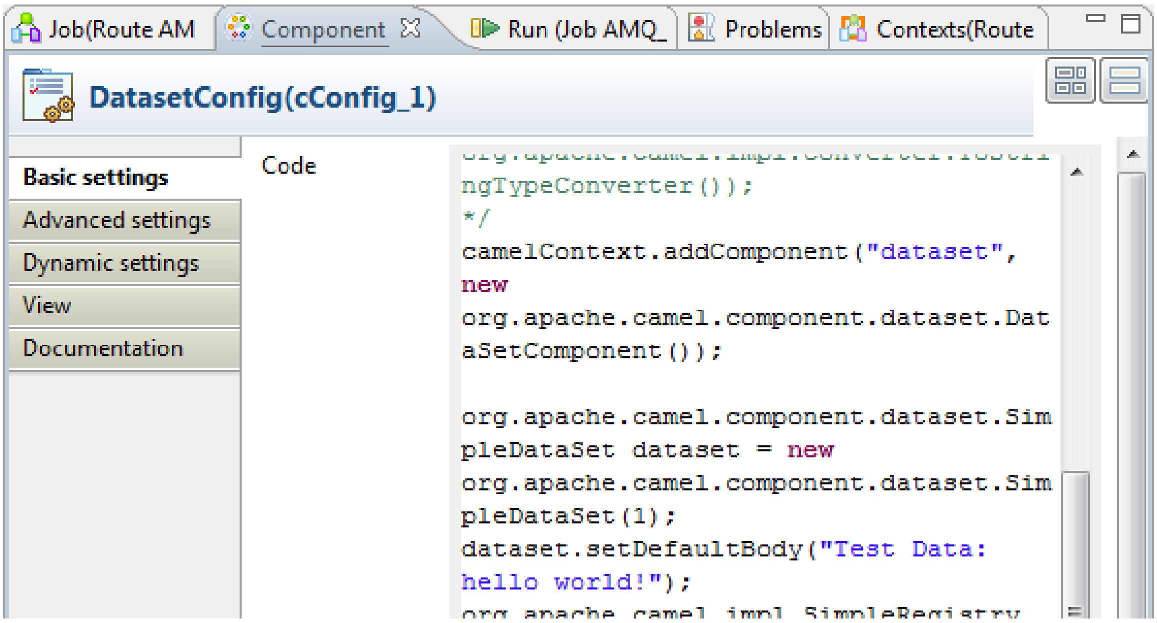
MQ Server: ActiveMQ * □ Use Transaction
Broker URI: "tcp://localhost:61616" *
□ Use PooledConnectionFacotry

2. From the **MQ Server** list, select an MQ server. In this use case, we use the default ActiveMQ server to handle the messages.

3. In the **Broker URI** field, type in Active MQ's default URI of the localhost server: *"tcp://localhost:61616"*.

> When using ActiveMQ to handle messages between different Routes, you need to launch the ActiveMQ server before executing the Routes. For more information on installing and launching ActiveMQ server, see the section about installing Apache ActiveMQ in the *Talend ESB Installation Guide*.

4. Double-click the **cConfig** component, which is labelled *DatasetConfig*, to display its **Basic settings** view in the **Component** tab and set its parameters.



Code

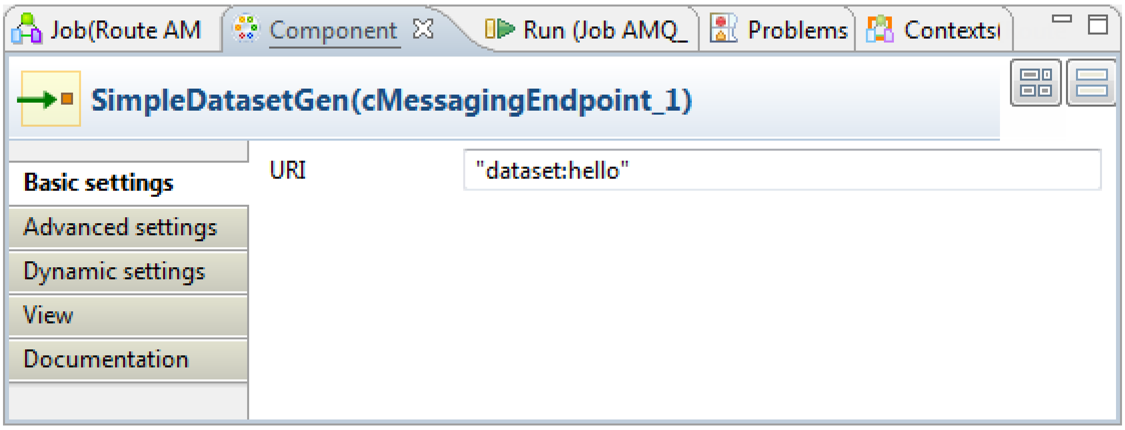
```
ngTypeConverter());
*/
camelContext.addComponent("dataset",
new
org.apache.camel.component.dataset.Dat
aSetComponent());

org.apache.camel.component.dataset.Sim
pleDataSet dataset = new
org.apache.camel.component.dataset.Sim
pleDataSet(1);
dataset.setDefaultBody("Test Data:
hello world!");
```

5. Write a piece of code in the **Code** field to register the dataset instance *hello* into the registry, as shown below.

```
org.apache.camel.component.dataset.SimpleDataSet dataset = new
org.apache.camel.component.dataset.SimpleDataSet(1);
dataset.setDefaultBody("Test Data: hello world!");
org.apache.camel.impl.SimpleRegistry registry = new
org.apache.camel.impl.SimpleRegistry();
     registry.put("hello",dataset);
```

```
camelContext.setRegistry(registry);
```

6. Double-click the **cMessagingEndpoint** component, which is labelled *SimpleDatasetGen*, to display its **Basic settings** view in the **Component** tab. and set its parameters.

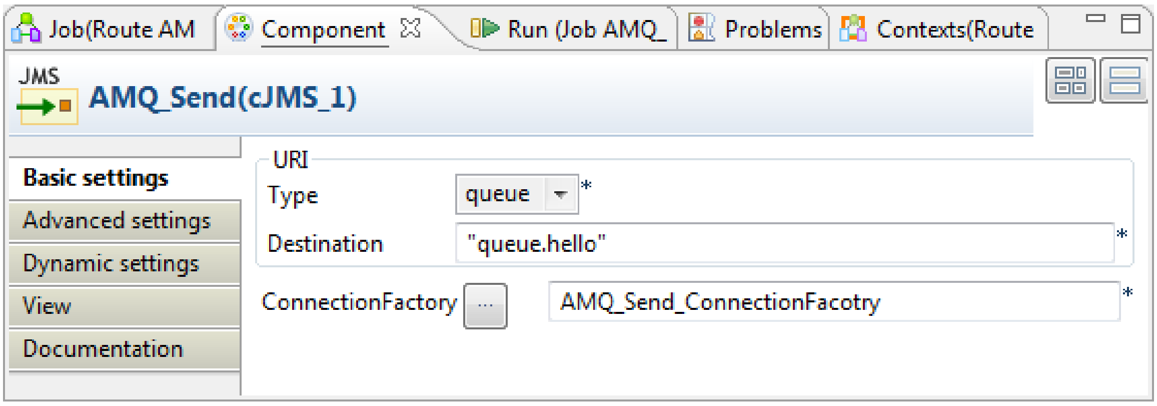


SimpleDatasetGen(cMessagingEndpoint_1)

URI "dataset:hello"

7. In the **URI** field, enter *dataset:hello* between the quotation marks.

8. Double-click the **cJMS** component labeled *AMQ_Send* to display its **Basic settings** view.



AMQ_Send(cJMS_1)

URI

Type queue *

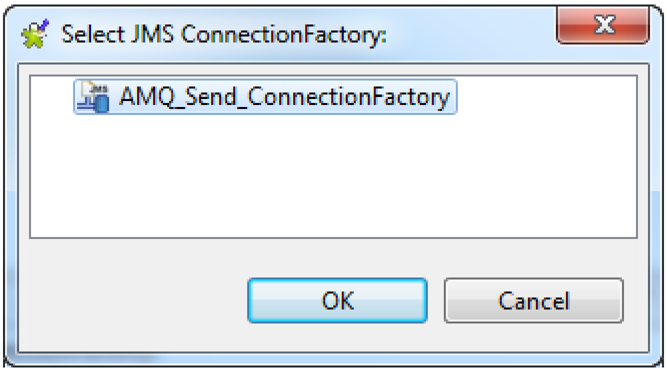
Destination "queue.hello" *

ConnectionFactory AMQ_Send_ConnectionFacotry *

9. From the **Type** list, select **queue** to send the message to a JMS queue.

   In the **Destination** field, type in a name for the JMS queue, *"queue.hello"* in this use case.

   Double-click the **[...]** button next to **ConnectionFactory**. Select the JMS connection factory that you have just configured in the dialog box and click **OK**. You can also enter the name of the **cJMSConnectionFactory** component directly in the field.



AMQ_Send_ConnectionFactory

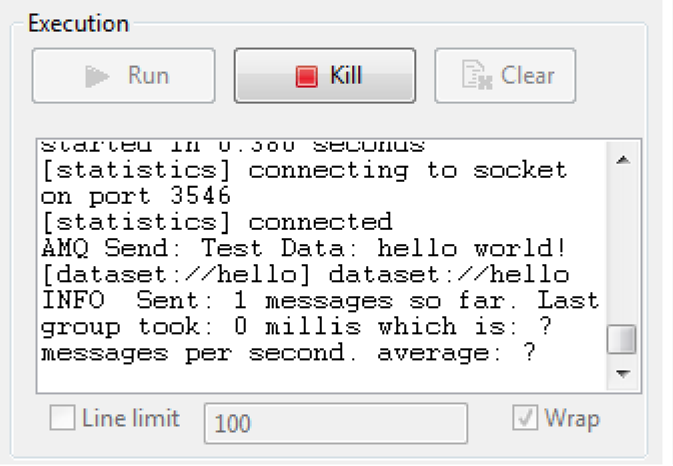
10. Double-click the **cProcessor** component labelled *PrintSendMsg* to display its **Basic settings** view in the **Component** tab, and customize the code in the **Code** area to display the sent message intercepted on the console.

```
System.out.println("AMQ Send: "+
exchange.getIn().getBody(String.class));
```

Procedure 3. Executing the Route

- Click the **Run** button in the **Run** view to launch the execution of your Route. You can also press **F6** to execute it.

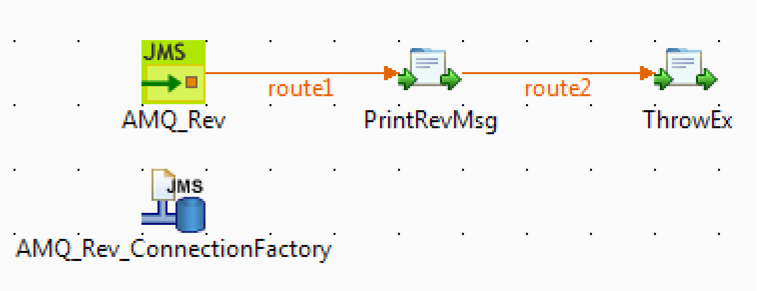
  RESULT: One *"hello world!"* message is sent to the JMS Queue, as shown in the **Run** console.



```
started in 0.380 seconds
[statistics] connecting to socket
on port 3546
[statistics] connected
AMQ Send: Test Data: hello world!
[dataset://hello] dataset://hello
INFO  Sent: 1 messages so far. Last
group took: 0 millis which is: ?
messages per second. average: ?
```

□ Line limit 100 ☑ Wrap

Testing the received message



Procedure 4. Dropping and linking the components

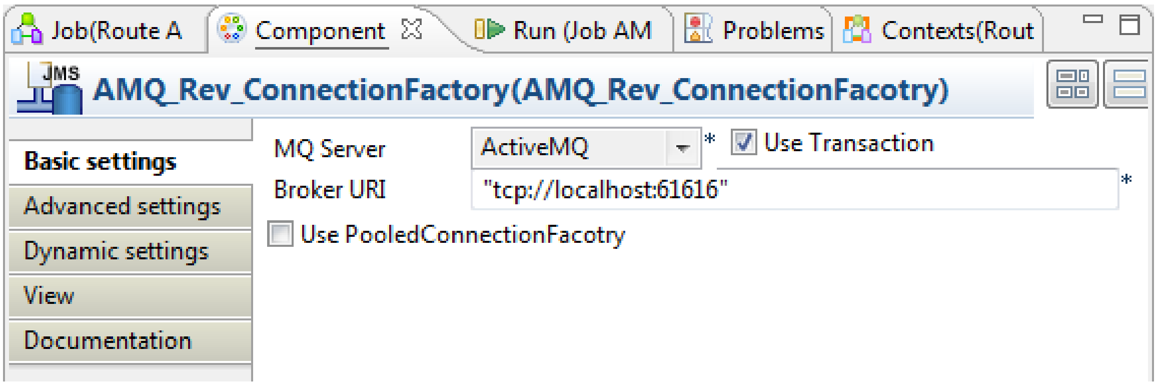
1. From the **Palette**, drop the four following components onto the design workspace: one **cJMS**, two **cProcessor** components and one **cJMSConnectionFactory**.

2. Connect the **cJMS** component to the first **cProcessor** using a **Row** > **Route** connection.

3. Connect the first **cProcessor** component to the second **cProcessor** component using a **Row** > **Route** connection.

Procedure 5. Configuring the components

1. Double-click the **cJMSConnectionFactory** component labelled *AMQ_Rev_ConnectionFactory* to display its **Basic settings** view in the **Component** tab.



AMQ_Rev_ConnectionFactory(AMQ_Rev_ConnectionFacotry)

MQ Server: ActiveMQ * ☑ Use Transaction

Broker URI: "tcp://localhost:61616" *

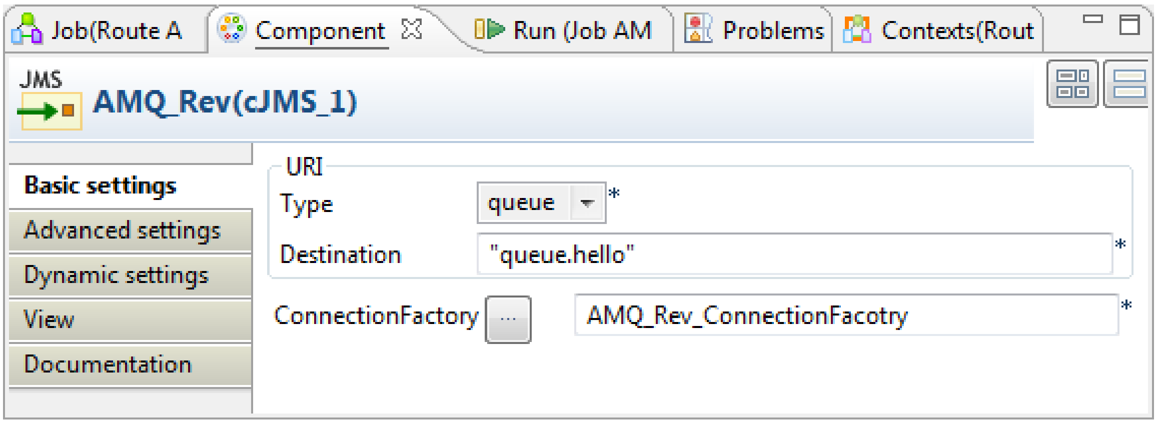
☐ Use PooledConnectionFacotry

2. From the **MQ Server** list, select an MQ server. In this use case, we use the default ActiveMQ server to handle the messages.

   Select the **Use transaction** check box.

3. In the **Broker URI** field, type in Active MQ's default URI of the localhost server: *"tcp://localhost:61616"*.

4. Double-click the **cJMS** component labeled *AMQ_Rev* to display its **Basic settings** view.



AMQ_Rev(cJMS_1)

URI

Type: queue *

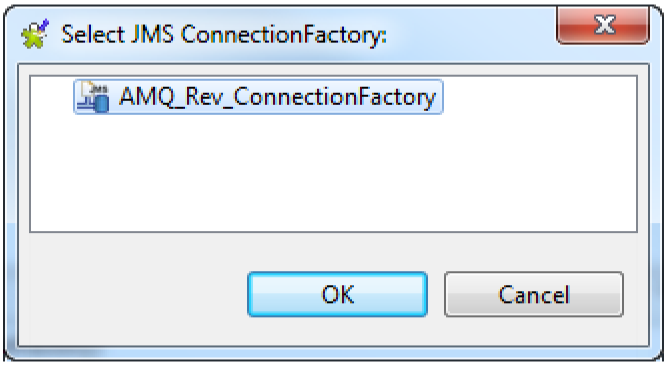
Destination: "queue.hello" *

ConnectionFactory [...] AMQ_Rev_ConnectionFacotry *

5. From the **Type** list, select **queue** to send the messages to a JMS queue.

   In the **Destination** field, type in a name for the JMS queue, *"queue.hello"* in this use case.

   Double-click the **[...]** button next to **ConnectionFactory**. Select the JMS connection factory that you have just configured in the dialog box and click **OK**. You can also enter the name of the **cJMSConnectionFactory** component directly in the field.



Select JMS ConnectionFactory:

AMQ_Rev_ConnectionFactory

6. Double-click the first **cProcessor** component labelled *PrintRevMsg* to display its **Basic settings** view in the **Component** tab, and customize the code in the **Code** area to display the received message intercepted on the console.

```
System.out.println("AMQ Receive: "+
exchange.getIn().getBody(String.class));
```

7. Double-click the second **cProcessor** component labelled *ThrowEx* to display its **Basic settings** view in the **Component** tab, and customize the code in the **Code** area to throw the *Force fail* exception every time an exchange is processed by the route.

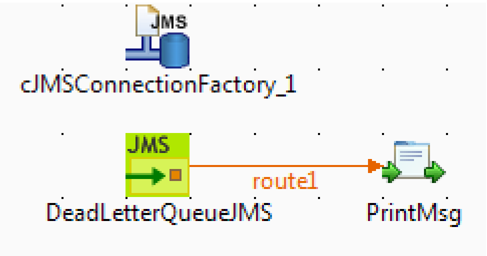
```
throw new Exception("Force fail")
```

Procedure 6. Executing the Route

- Click the **Run** button in the **Run** view to launch the execution of your Route. You can also press **F6** to execute it.

  RESULT: The *"hello world!"* message is tested and a rollback transaction is performed. Once the message redelivery attempts exceeds six times, the pending message is sent to the *Dead Letter* JMS Queue.

Consuming the message from the *DeadLetter* JMS queue



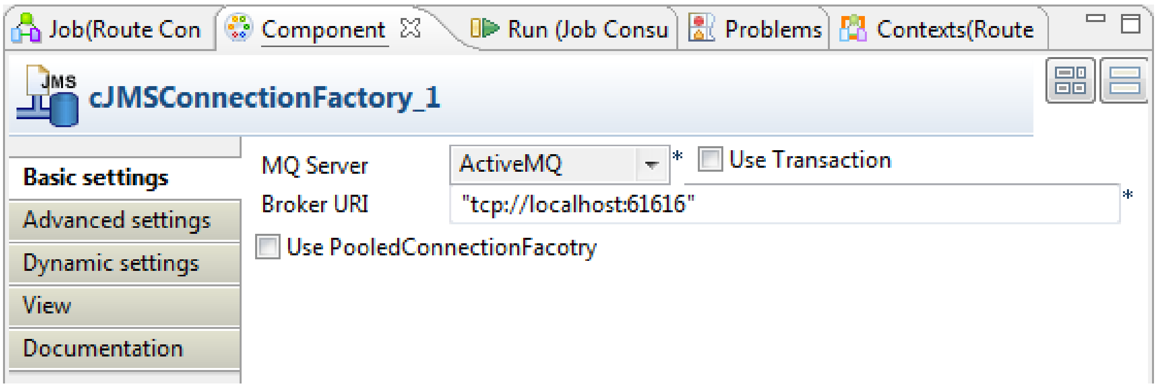
Procedure 7. Dropping and linking the components

1. From the **Palette**, drop the three following components onto the design workspace: one **cJMSConnectionFactory**, one **cJMS** and one **cProcessor** component.

2. Connect the **cJMS** component to the **cProcessor** component using a **Row** > **Route** connection.

Procedure 8. Configuring the components

1. Double-click the **cJMSConnectionFactory** component to display its **Basic settings** view in the **Component** tab.



MQ Server: ActiveMQ * □ Use Transaction

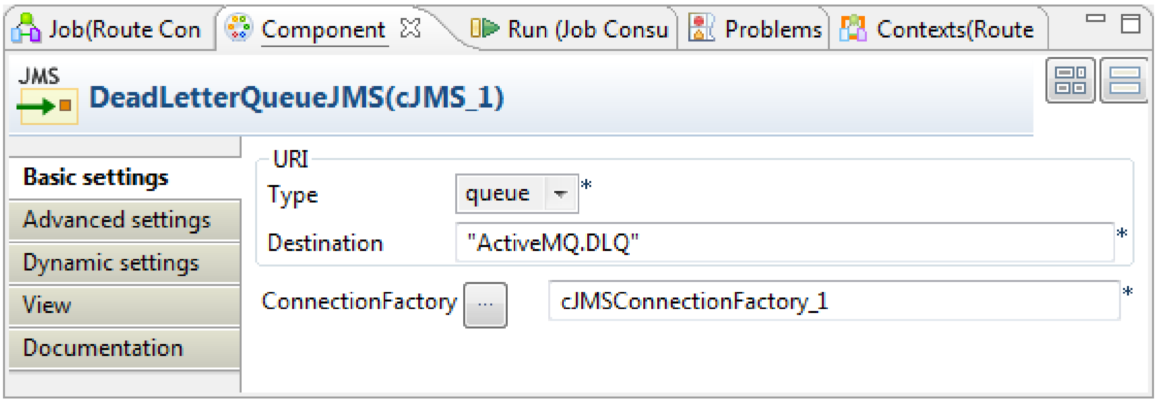
Broker URI: "tcp://localhost:61616" *

□ Use PooledConnectionFacotry

2. From the **MQ Server** list, select an MQ server. In this use case, we use the default ActiveMQ server to handle the messages.

3. In the **Broker URI** field, type in Active MQ's default URI of the localhost server: *"tcp://localhost:61616"*.

4. Double-click the **cJMS** component labeled *DeadLetterQueueJMS* to display its **Basic settings** view.



Type: queue *

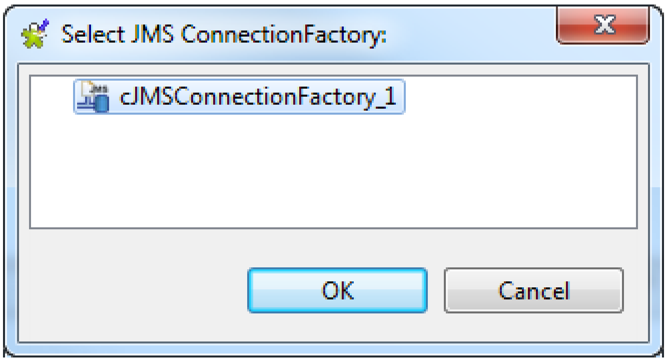
Destination: "ActiveMQ.DLQ" *

ConnectionFactory: cJMSConnectionFactory_1 *

5. From the **Type** list, select **queue** to send the messages to a JMS queue.

   In the **Destination** field, type in a name for the JMS queue, *"ActiveMQ.DLQ"* in this use case (the default *Dead Letter* Queue in ActiveMQ).

   Double-click the **[...]** button next to **ConnectionFactory**. Select the JMS connection factory that you have just configured in the dialog box and click **OK**. You can also enter the name of the **cJMSConnectionFactory** component directly in the field.



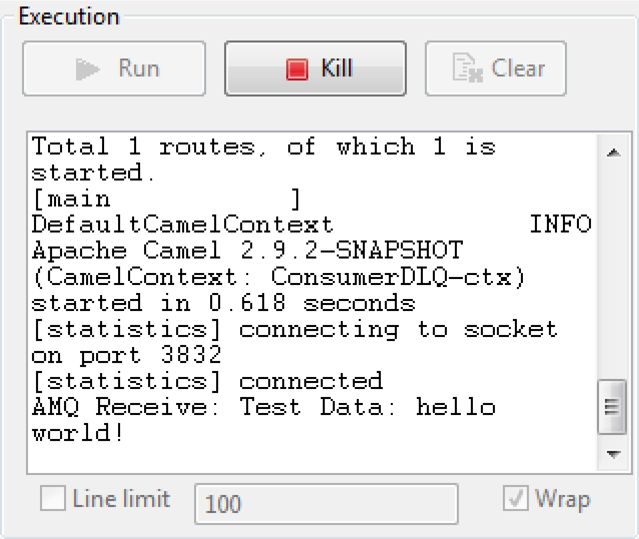
6. Double-click the **cProcessor** component labelled *PrintMsg* to display its **Basic settings** view in the **Component** tab, and customize the code in the **Code** area to display the received message intercepted on the console.

```
System.out.println("AMQ Receive: "+
exchange.getIn().getBody(String.class));
```

Procedure 9. Executing the Route

- Click the **Run** button in the **Run** view to launch the execution of your Route. You can also press **F6** to execute it.

  RESULT: The *"hello world!"* message that was in the *Dead Letter* queue is consumed, as shown in the **Run** console.



```
Total 1 routes, of which 1 is
started.
[main              ]
DefaultCamelContext             INFO
Apache Camel 2.9.2-SNAPSHOT
(CamelContext: ConsumerDLQ-ctx)
started in 0.618 seconds
[statistics] connecting to socket
on port 3832
[statistics] connected
AMQ Receive: Test Data: hello
world!
```

cMail

cMail Properties

Component family	Messaging	
Function	**cMail** is designed to send or receive mails.	
Purpose	Sends or receives mails in a route.	
Basic settings	*Protocols*	List of protocols for sending or receiving mails.
	Host	Host name of the mail server.
	Port	Port number of the mail server.
	UserName and *Password*	Login authentication data.
	Subject	Subject of the mail being sent.
	Content Type	The mail content type.
	From	The mail sender.
	To	The mail receivers.
	CC	The CC recipients of the mail. Separate multiple email addresses with a comma.
	BCC	The BCC recipients of the mail. Separate multiple email addresses with a comma.
Advanced settings	*Arguments*	Click the **[+]** button to add lines as needed in the **Arguments** table. Then, enter the name and value of an argument.
Usage	When used as a start component, **cMail** is intended to receive mails. Otherwise, it is intended to send mails.	
Limitation	n/a	

Scenario: Using cMail to send and receive mails

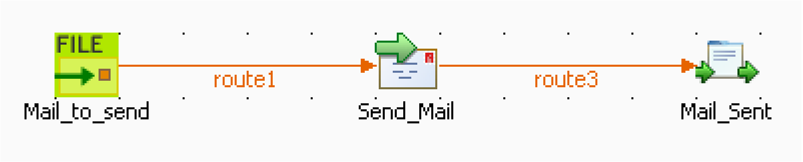
This scenario includes two routes. The first one sends a mail while the second receives it.

Now we build a route to send a mail.

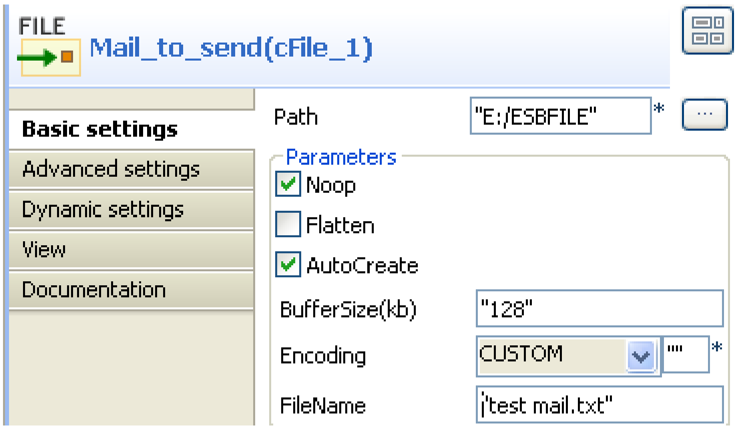
Procedure 10. Mail sending

1. Drop the components from the **Palette** onto the workspace: **cFile**, **cMail** and **cProcessor**, respectively labelled as **Mail_to_send**, **Send_Mail** and **Mail_Sent**.

2. Link the components using a **Row** > **Route** connection.



3. Double-click **cFile** to open its **Basic settings** view in the **Component** tab.



Mail_to_send(cFile_1)

Path "E:/ESBFILE" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

Encoding CUSTOM "" *

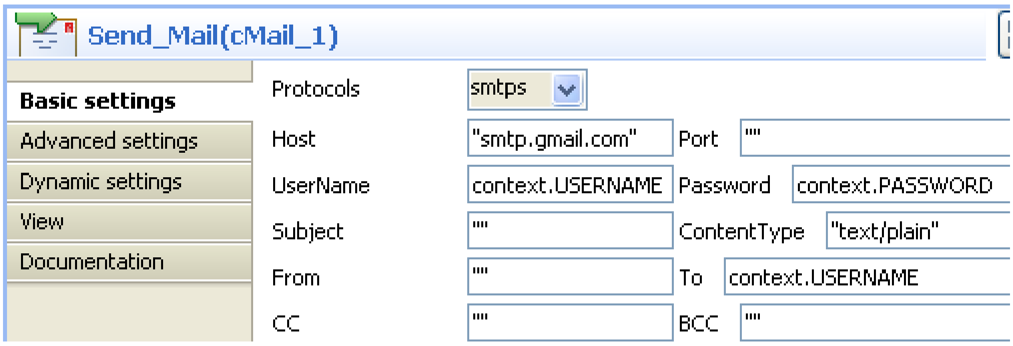
FileName "test mail.txt"

4. Click the **[...]** button next to the **Path** field to select the folder that has the file to send.

5. In the **FileName** field, enter the name of the file to send, *test mail.txt* in this use case. Keep the default setup of other items.

   The content of this file is *test mail body*.

6. Double-click **cMail** to open its **Basic settings** view in the **Component** tab.



Send_Mail(cMail_1)

Protocols smtps

Host "smtp.gmail.com" Port ""

UserName context.USERNAME Password context.PASSWORD

Subject "" ContentType "text/plain"

From "" To context.USERNAME

CC "" BCC ""

7. In the **Protocols** list, select *smtps*.

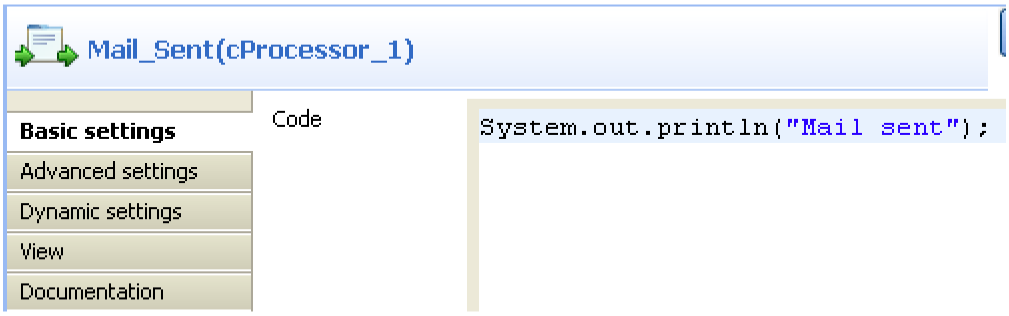
   In the **Host** field, type in the host name of the smtp server, *smtp.gmail.com* in this use case.

   In the **UserName** and **Password** fields, enter the login authentication credentials, which are in the form of context variables in this example. For more information about context variable setup, see *Talend Open Studio for ESB* **User Guide**.

   Keep the default setting of the **ContentType** field, i.e. *text/plain*.

   In the **To** field, enter the receiver of the mail, which is also in the form of context variable in this example.

8. Double-click **cProcessor** to open its **Basic settings** view in the **Component** tab.

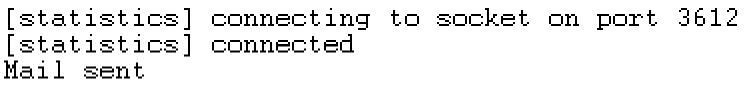


Code
```
System.out.println("Mail sent");
```

9. In the **Code** box, enter the code below to give a prompt after the mail is sent.

```
System.out.println("Mail sent");
```

10. Save the route and press **F6** to run.



```
[statistics] connecting to socket on port 3612
[statistics] connected
Mail sent
```

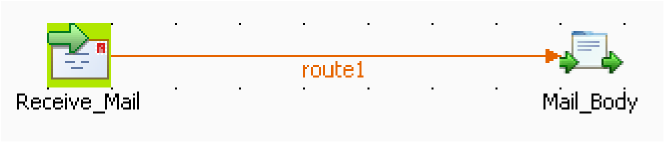
As shown above, the mail has been sent out successfully.

Now we build a route to receive the mail.

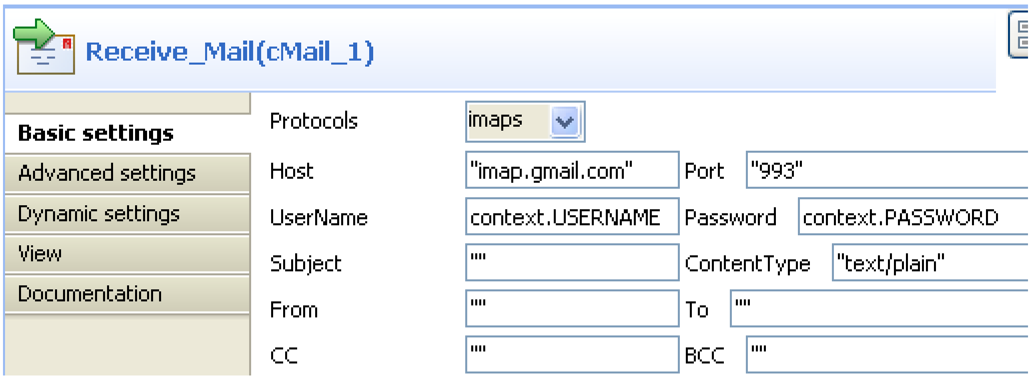
Procedure 11. Mail receiving

1. Drop the components from the **Palette** onto the workspace: **cMail** and **cProcessor**, respectively labelled as **Receive_Mail** and **Mail_Body**.

2. Link the components using a **Row** > **Route** connection.



3. Double-click **cMail** to open its **Basic settings** view in the **Component** tab.



Protocols: imaps
Host: "imap.gmail.com" Port: "993"
UserName: context.USERNAME Password: context.PASSWORD
Subject: "" ContentType: "text/plain"
From: "" To: ""
CC: "" BCC: ""

4. In the **Protocols** list, select *imaps*.

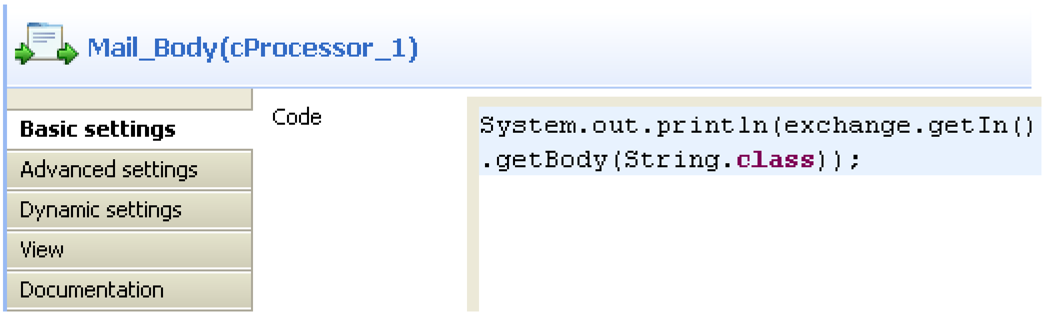
5. In the **Host** field, type in the host name of the imap server, *imap.gmail.com* in this use case.

6. In the **Port** field, type in the port number, *993* in this use case.

7. In the **UserName** and **Password** fields, enter the login authentication credentials, which are in the form of context variables in this example. For more information about context variable setup, see *Talend Open Studio for ESB* **User Guide**.

8. Keep the default setting of the **ContentType** field, i.e. *text/plain*.

9. Double-click **cProcessor** to open its **Basic settings** view in the **Component** tab.

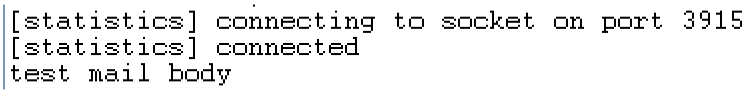


```
System.out.println(exchange.getIn()
.getBody(String.class));
```

10. In the **Code** box, enter the code below to print the mail body.

```
System.out.println(exchange.getIn().getBody(String.class));
```

11. Save the route and press **F6** to run.



```
[statistics] connecting to socket on port 3915
[statistics] connected
test mail body
```

As shown above, the mail has been received and its content is *test mail body*.

cMessagingEndpoint

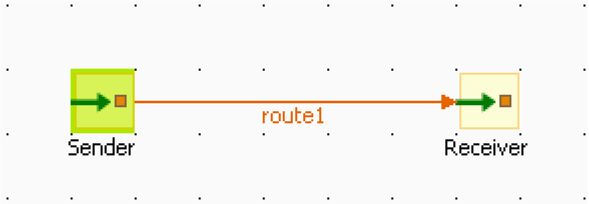
cMessagingEndpoint properties

Component Family	Messaging	
Function	**cMessagingEndpoint** allows two applications to communicate by either sending or receiving messages, one endpoint can not do both.	
Purpose	**cMessagingEndpoint** sends or receives messages.	
Basic settings	*URI*	URI of the messages to send or receive. It can be of different format: -File: "file:/", -Database: "jdbc:/", -Protocols: "ftp:/", "http:/" -etc. You can add parameters to the URI using the generic URI syntax, for example: `"file:/directoryName?option=value&option=value"` For more information on the different components that can be used in cMessagingEndpoint, see Apache Camel's Website: http://camel.apache.org/components.html.
Advanced settings	*Dependencies*	By default, the camel core supports the following components: bean, browse, class, dataset, direct, file, language, log, mock, properties, ref, seda, timer, vm. To use other components, you have to provide the dependencies corresponding to those components in the **cMessagingEndpoint** component. To do so: Click the plus button to add new lines in the **Camel component** list. In the line added, select the component you want to use in **cMessagingEndpoint**.
	Use a custom component	If you want to use a custom component, select this check box and click the three-dot button to upload a jar file with your own component. All the transitive dependencies of this custom component should be included in the jar file.
Usage	This component can be used as sending and/or receiving message endpoint according to its position in the Route.	

Limitation	n/a

Scenario 1: Moving files from one message endpoint to another

This scenatio uses two **cMessagingEndpoint** components to read and move files from one endpoint to another.



Dropping and linking the components

1. From the **Messaging** folder of the **Palette**, drag and drop two **cMessagingEndpoint** components onto the design workspace, one as the message sender and the other as the message receiver, and label them *Sender* and *Receiver* respectively to better identify their roles in the Route.

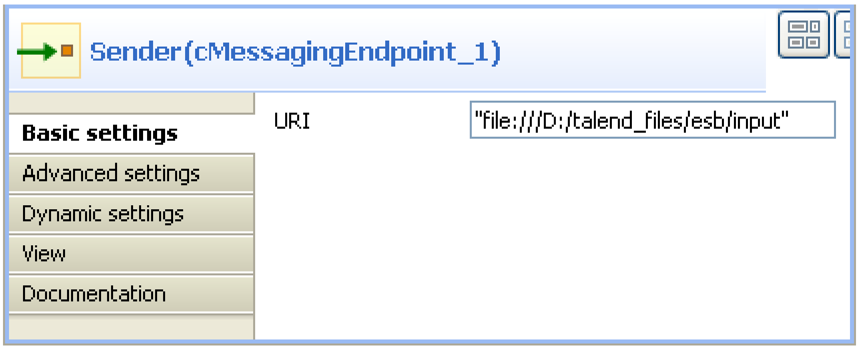
2. Right-click the component labeled *Sender*, select **Row** > **Route** in the menu and drag to the *Receiver* to link them together with a route link.

Configuring the components and connections

1. Double-click the component labeled *Sender* to open its **Basic settings** view in the **Component** tab.

2. In the **URI** field, type in the URI of the messages you want to route.

   As we are handling files, type in *"file:///"* and the path to the folder containing the files.

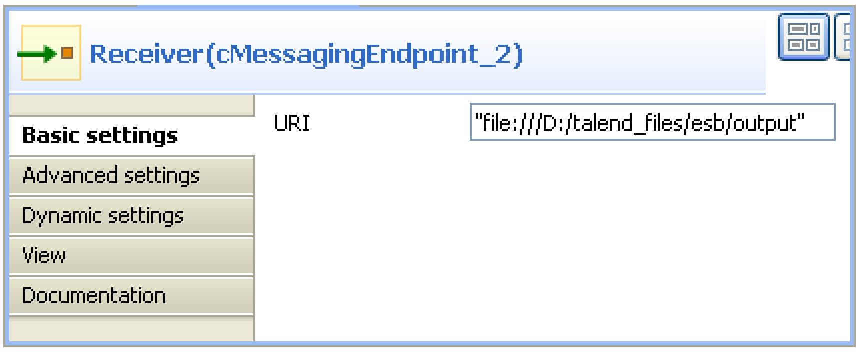


URI "file:///D:/talend_files/esb/input"

3. Double-click the component labeled *Receiver* to open its **Basic settings** view in the **Component** tab.

4. In the **URI** field, type in the URI of the folder where you want to route your message.

   As we are handling files, type in *"file:///"* and the path to the folder to which the files will be sent.



Receiver(cMessagingEndpoint_2)

URI "file:///D:/talend_files/esb/output"

5. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. To have a look at the generated code, click the **Code** tab at the bottom of the design workspace.

```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender")).routeId("Sender").to(
                    uriMap.get("Receiver")).id(
                    "cMessagingEndpoint_2");
        }
    };
    getCamelContexts().get(0).addRoutes(routeBuilder);
}
```

The code shows the `from` and `.to` corresponding to the two endpoints: `from` for the sending one and `.to` for the receiving one.

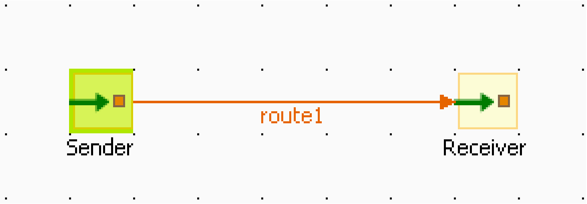
2. In the **Run** view, click the **Run** button to launch the execution of your Route.

You can also press **F6** to execute it.

RESULT: The files are moved from their original folder to the target one. Furthermore, a new **.camel** folder is created in the source folder containing the consumed files. This is Camel's default behavior. Thus, the files will not be processed endlessly but they are backed up in case of problems.

Scenario 2: sending files to another message endpoint

This scenario accesses FTP service and transfers files from one endpoint to another.



Dropping and linking components

1. From the **Messaging** folder of the **Palette**, drag and drop two **cMessagingEndpoint** components onto the design workspace, one as the message sender and the other as the message receiver, and label them *Sender* and *Receiver* respectively to better identify their roles in the Route.

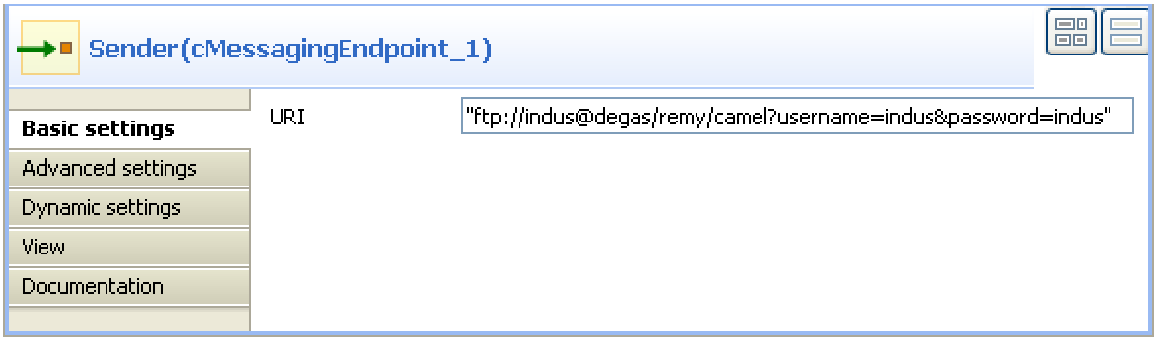
2. Right-click the component labeled *Sender*, select **Row** > **Route** in the menu and drag to the *Receiver* to link them together with a route link.

Configuring the components and connections

1. Double-click the component labeled *Sender* to display its **Basic settings** view in the **Component** tab.

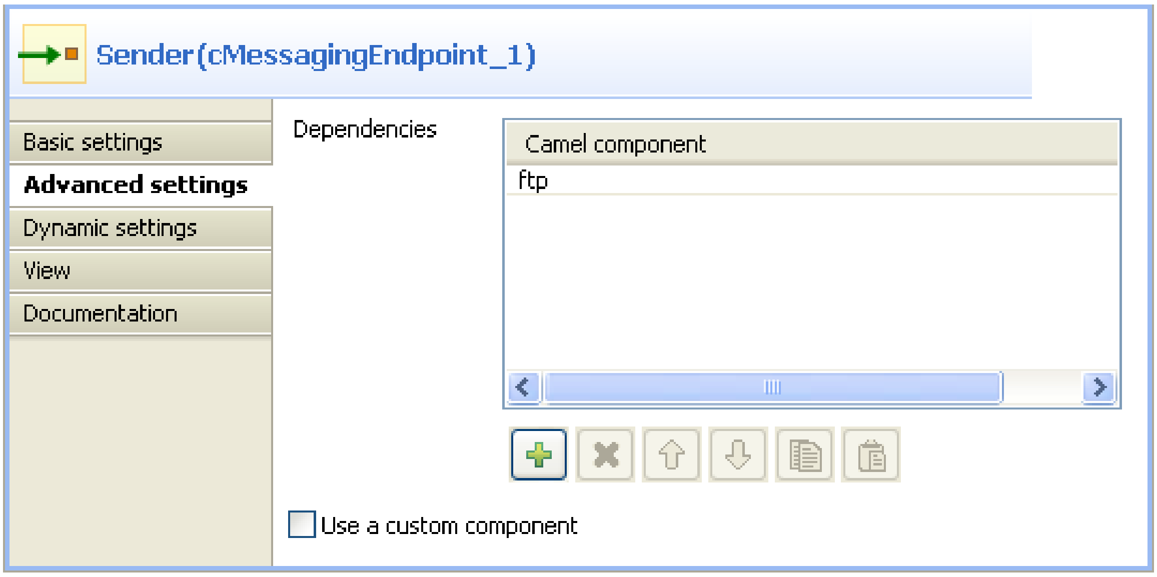
2. In the **URI** field, type in the URI of the message you want to route.

   Here, we are using an FTP component: `ftp://indus@degas/remy/camel` with URI specific parameters authenticating the FTP connection: `?username=indus&password=indus`.



URI "ftp://indus@degas/remy/camel?username=indus&password=indus"

3. For the FTP component to work in Camel, click the **Advanced settings** tab of **cMessagingEndpoint**, click the **[+]** button to add a Camel component in the **Dependencies** table, and select *ftp* from the **Camel component** list to activate the FTP component.



Dependencies

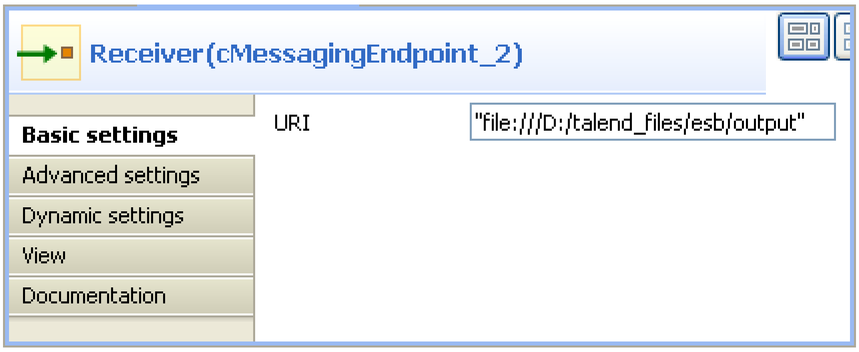
Camel component
ftp

□ Use a custom component

4. Double-click the component labeled *Receiver* to open its **Basic settings** view in the **Component** tab.

5. In the **URI** field, type in the URI of the folder to which you want your message to be routed.

   As we are handling files, type in *"file:///"* and the path to the folder to which the files will be sent.

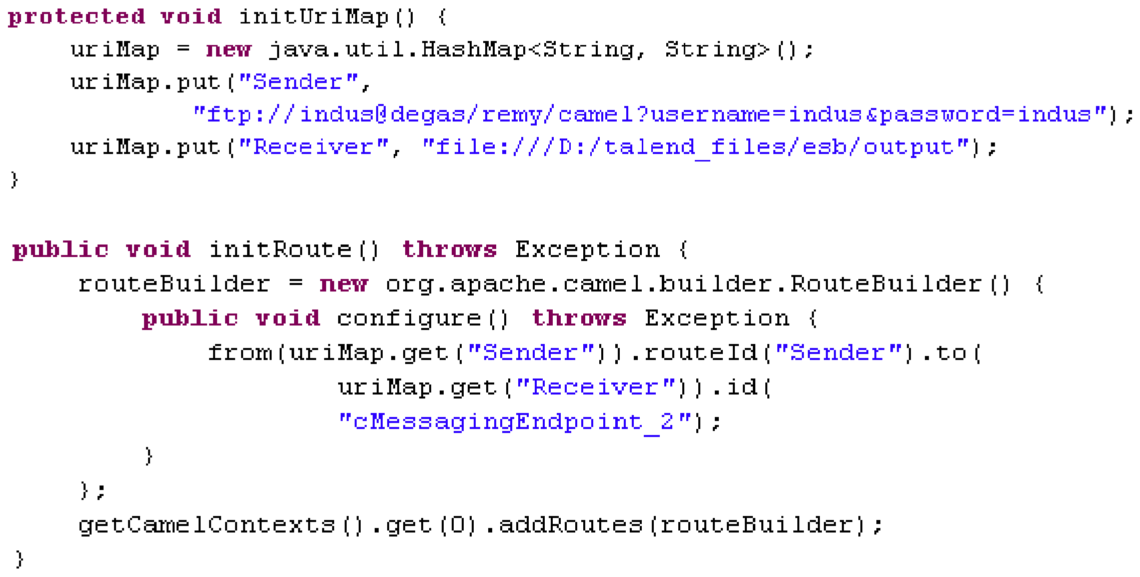


URI "file:///D:/talend_files/esb/output"

6. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. To have a look at the generated code, click the **Code** tab at the bottom of the design workspace.



```
protected void initUriMap() {
    uriMap = new java.util.HashMap<String, String>();
    uriMap.put("Sender",
            "ftp://indus@degas/remy/camel?username=indus&password=indus");
    uriMap.put("Receiver", "file:///D:/talend_files/esb/output");
}

public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender")).routeId("Sender").to(
                    uriMap.get("Receiver")).id(
                    "cMessagingEndpoint_2");
        }
    };
    getCamelContexts().get(0).addRoutes(routeBuilder);
}
```

In this part of code, we can see a route represented by `from` and `.to`, corresponding to the sending and receiving endpoints.

2. In the **Run** view, click the **Run** button to launch the execution of your Route.

You can also press **F6** to execute it.

RESULT: The message is sent (copied) to the receiving endpoint.

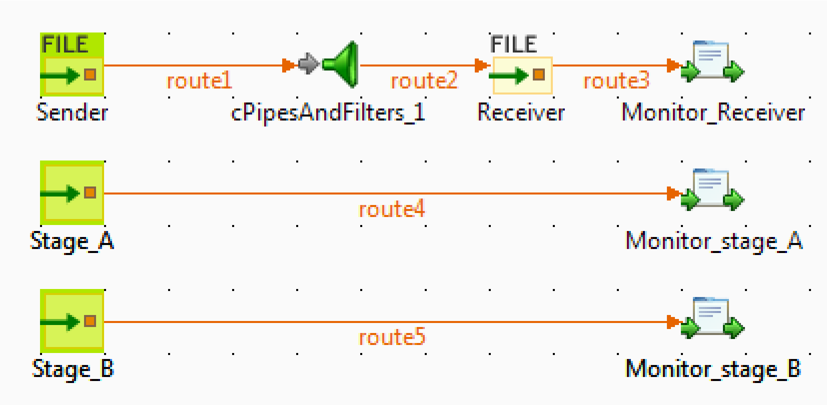
cPipesAndFilters

cPipesAndFilters properties

Component Family	Messaging	
Function	The **cPipesAndFilters** component divides message processing into a sequence of independent endpoint instances, which can then be chained together.	
Purpose	This component allows you to split message routing into a series of independent processing stages.	
Basic settings	*URI list*	Click the plus button to add new lines for URIs that identify endpoints.
Usage	**cPipesAndFilters** is usually used in the middle of a Route.	
Limitation	n/a	

Scenario: Using cPipesAndFilters to process the task in sequence

In this scenario, a **cPipesAndFilters** component is used so that messages sent from the sender endpoint undergo stage A and stage B. Upon completion of both stages, the messages are routed to a file system, which is the receiver endpoint for the messages.



Dropping and linking the components

1. From the **Messaging** folder of the **Palette**, drop two **cFile** components onto the design workspace, one as the message sender and the other as the message receiver, and label them *Sender* and *Receiver* respectively to better identify their roles in the Route.

2. From the **Messaging** folder, drop one **cPipesAndFilters** component onto the design workspace, between the two **cFile** components.

3. From the **Messaging** folder, drop two **cMessagingEndpoint** components onto the design workspace, one as the endpoint of stage A and the other as the endpoint of stage B, and label them *Stage_A* and *Stage_B* respectively to better identify their roles in the Route.

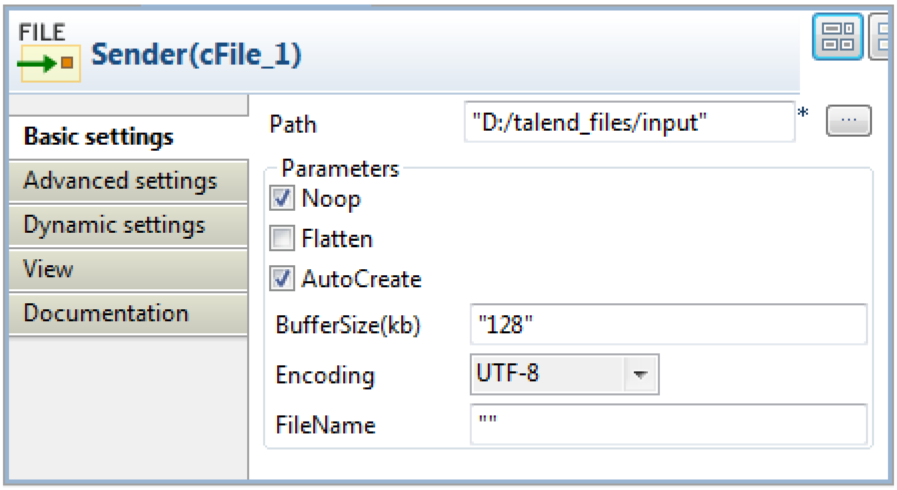
4. From the **Processor** folder, drop three **cProcessor** components onto the design workspace to monitor messages received on the receiver, stage A and stage B endpoints respectively, and label them *Monitor_Receiver*, *Monitor_stage_A*, and *Monitor_stage_B* respectively to better identify their roles in the Route.

5. Right-click the **cFile** component labeled *Sender*, select **Row** > **Route** from the contextual menu, and click the **cPipesAndFilters** component.

   Repeat this step to set up the rest **Row** > **Route** connections, as shown above.

Configuring the components

1. Double-click the **cFile** component labeled *Sender* to open its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/input" *

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

Encoding UTF-8

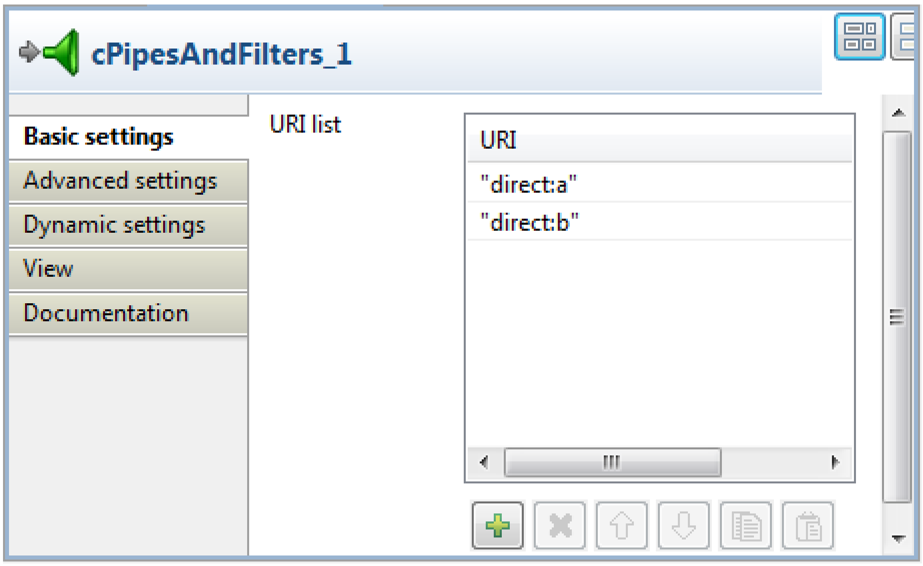
FileName ""

2. In the **Path** field, fill in or browse to the path to the folder that holds the source files.

3. From the **Encoding** list, select the encoding type of your source files. Leave the other parameters as they are.

4. Repeat these steps to define the path to the output files and the output encoding type in the **Basic settings** view of the **cFile** component labeled *Receiver*.

5. Double-click the **cPipesAndFilters** component to open its **Basic settings** view in the **Component** tab.



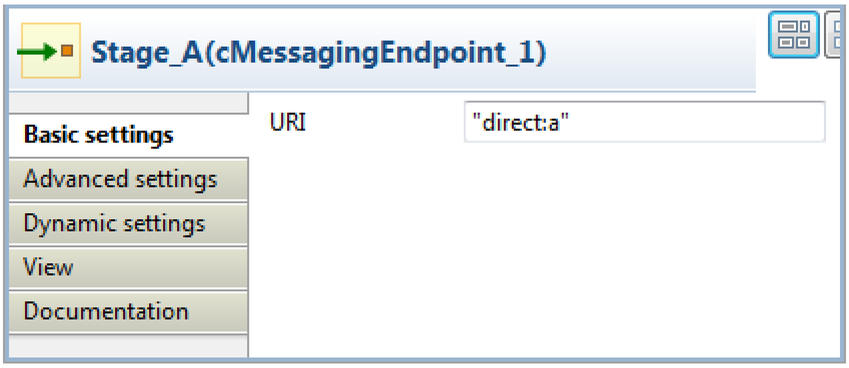
cPipesAndFilters_1

URI list

URI
"direct:a"
"direct:b"

6. Click the plus button to add two lines to the **URI list** table, and fill the first line with `"direct:a"` and the second line with `"direct:b"` to define the URIs of stage A and stage B that the messages will undergo.

7. Double-click the **cMessagingEndpoint** component labeled *Stage_A* to configure the component in its **Basic settings** view and define the URI of stage A.



Stage_A(cMessagingEndpoint_1)

URI "direct:a"

Repeat this step to define the URI of stage B in the **Basic settings** view of the **cMessagingEndpoint** component labeled *Stage_B*.

8. Double-click the **cProcessor** component labeled *Monitor_Receiver* to open its **Basic settings** view, and customize the code in the **Code** area to display the file names of the messages received on Receiver, as follows:

```
System.out.println("Message sent to Receiver: "+
exchange.getIn().getHeader("CamelFileName"));
```

Repeat this step to customize the code in the other two **cProcessor** components to display the file names of the messages received on stage A and stage B respectively:

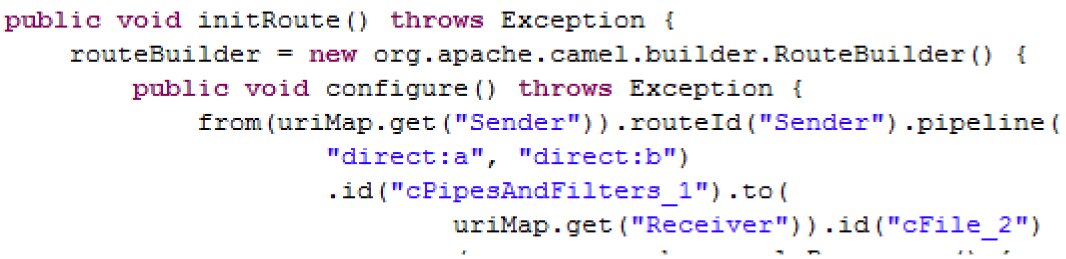
```
System.out.println("Message sent to stage A: "+
exchange.getIn().getHeader("CamelFileName"));
```

```
System.out.println("Message sent to stage B: "+
exchange.getIn().getHeader("CamelFileName"));
```

9. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

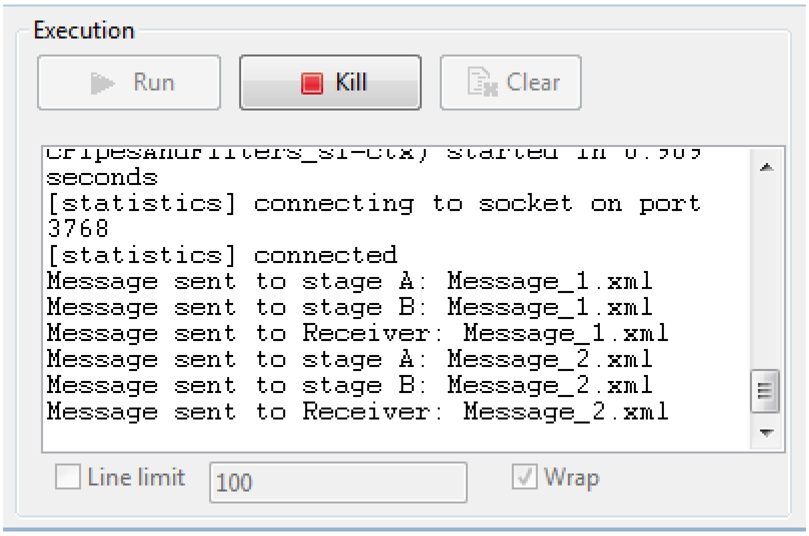


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender")).routeId("Sender").pipeline(
                    "direct:a", "direct:b")
                    .id("cPipesAndFilters_1").to(
                            uriMap.get("Receiver")).id("cFile_2")
```

As shown in the code, messages sent `from` `Sender` are redirected to endpoints identified by `direct:a` and `direct:b` by `cPipesAndFilters_1` before being routed to `Receiver`.

2. Press **F6** to run your Route.

RESULT: The message delivery goes through *stage A* and then *stage B* before reaching *Receiver*.



```
cPipesAndFilters_s1-ctx) started in 0.909
seconds
[statistics] connecting to socket on port
3768
[statistics] connected
Message sent to stage A: Message_1.xml
Message sent to stage B: Message_1.xml
Message sent to Receiver: Message_1.xml
Message sent to stage A: Message_2.xml
Message sent to stage B: Message_2.xml
Message sent to Receiver: Message_2.xml
```

□ Line limit 100 ☑ Wrap

cTalendJob

cTalendJob properties

Component Family	Messaging	
Function	**cTalendJob** allows you to import a library.	
Purpose	**cTalendJob** calls a **Talend** Job exported as **OSGI Bundle For ESB**. For more information on how to export a Job as OSGI Bundle, see *Talend Open Studio for ESB User Guide*.	
Basic settings	*Library*	Select the library you want to import from the list, or click on the [...] button to import the jar library of your **Talend** Job.
	Job	Type in the name of the package and the name of your job separated by a point. For example: *route_project.txmlmap_0_1.tXMLMap* To get this naming, you can open the jar library of your Job, go to OSGI-INF > blueprint and edit the job.xml file, the naming is available in a bean node like `<bean id="job" class="route_project.txmlmap_0_1.tXMLMap"/>`
	Context	Type in the name of the context to use to execute your Job
Usage	**cTalendJob** can be a start, middle or end component in a Route.	
Limitation	n/a	

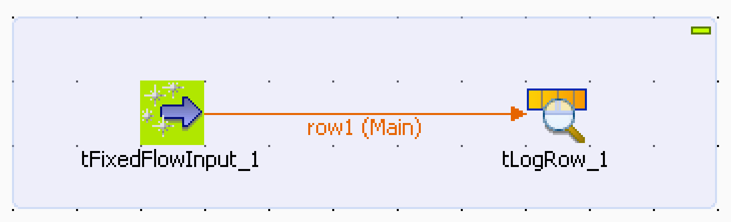
Scenario: Using camel message headers as context parameters to call a job

In this scenario, a Data Integration Job is built with a context variable defined in the **Integration** perspective. Then, a Route is established in the **Mediation** perspective with the message header defined the same as the context variable in the DI Job. Meanwhile, a **cTalendJob** component is deployed to call the DI Job and pass the value of the Route's message header to the DI Job's context variable.

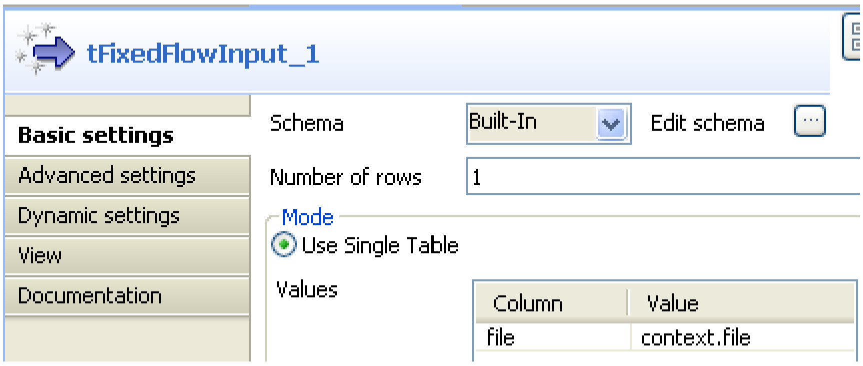
Building a DI Job and exporting it as an OSGI Bundle for ESB

1. In the **Integration** perspective, drop the following components from the **Palette** onto the workspace: **tFixedFlowInput** and **tLogRow**.

2. Link the components using a **Row** > **Main** connection.



3. Double-click **tFixedFlowInput** to open its **Basic settings** view in the **Component** tab.



tFixedFlowInput_1

Schema: Built-In   Edit schema [...]

Number of rows: 1

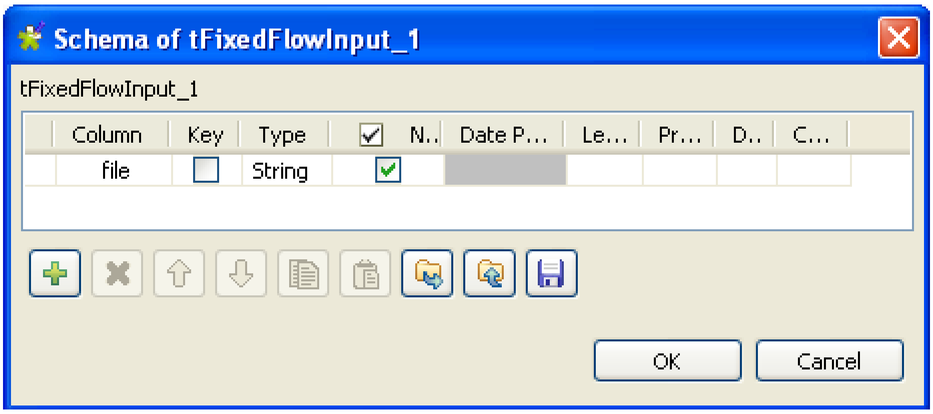
Mode

● Use Single Table

Values

Column	Value
file	context.file

4. Click the **[...]** button next to **Edit schema** to open the schema editor.



Schema of tFixedFlowInput_1

tFixedFlowInput_1

Column	Key	Type	☑ N..	Date P...	Le...	Pr...	D..	C...
file	□	String	☑					

Click the **[+]** button to add a line.

Enter *file* as the column name and choose *String* as the data type.

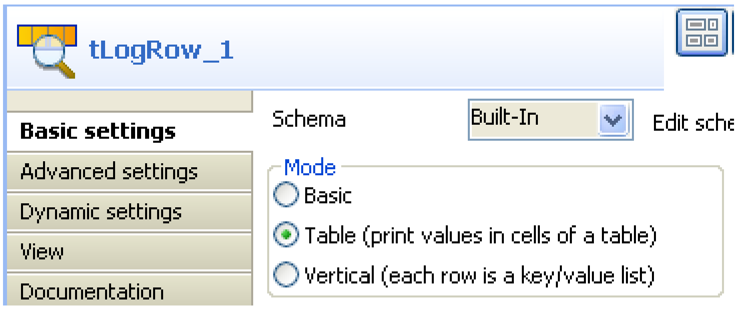
Click **OK** to close the editor.

5. Select the **Use Single Table** option and enter *context.file* as the value.

Note that the context *default* with the variable *file* has been defined.

For more information about the context setup, see *Talend Open Studio for ESB User Guide*.

6. Double-click **tLogRow** to open its **Basic settings** view in the **Component** tab.



tLogRow_1

Schema: Built-In   Edit sche

Mode

○ Basic

● Table (print values in cells of a table)

○ Vertical (each row is a key/value list)

7. Select **Table (print values in cells of a table** for a better display.

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

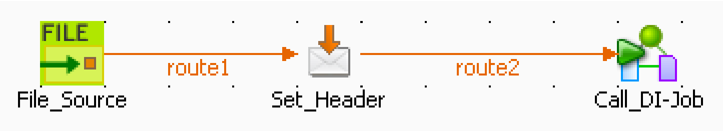
9. Export the Job as an *OSGI Bundle for ESB*.

10. Unzip the generated jar file.

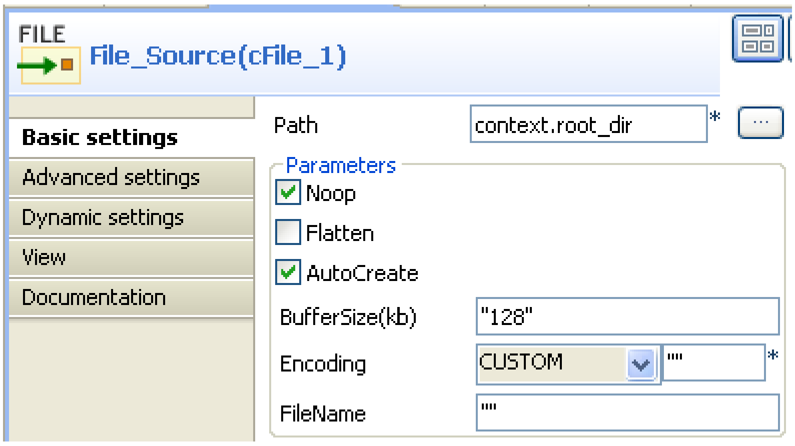
Building a Route for exchanging messages and calling the DI Job

1. In the **Mediation** perspective, drop the following components from the **Palette** onto the workspace: **cFile**, **cSetHeader** and **cTalendJob**, respectively labelled as **File_Source**, **Set_Header** and **Call_DI-Job**.

2. Link the components using a **Row** > **Route** connection.



3. Double-click **cFile** to open its **Basic settings** view in the **Component** tab.



Path: context.root_dir

Parameters

☑ Noop

☐ Flatten

☑ AutoCreate

BufferSize(kb): "128"

Encoding: CUSTOM ""

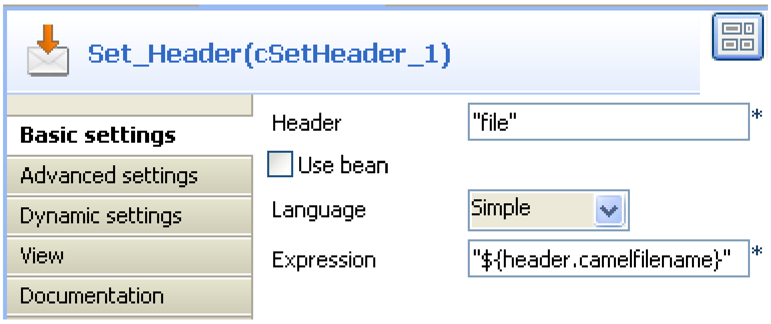
FileName: ""

4. In the **Path** field, enter the variable *context.root_dir* to specify the file path.

   Keep other default settings as they are.

   For more information about the context setup, see *Talend Open Studio for ESB User Guide*.

5. Double-click **cSetHeader** to open its **Basic settings** view in the **Component** tab.



Header: "file"

☐ Use bean

Language: Simple

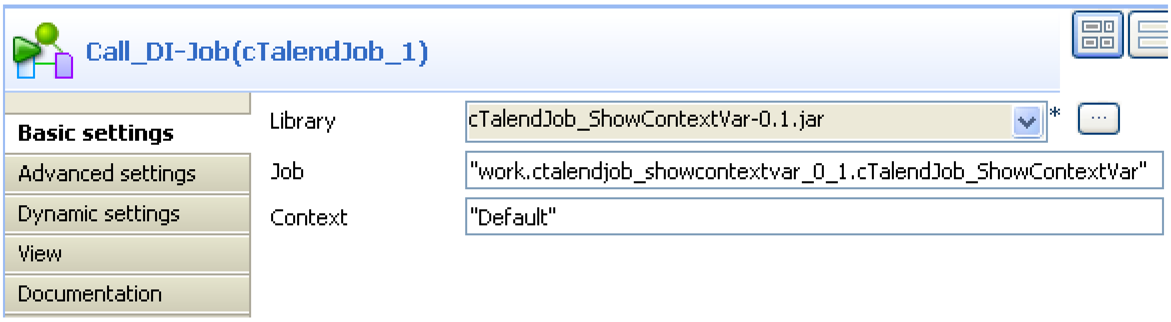
Expression: "${header.camelfilename}"

6. In the **Header** field, enter *file*, which is the same as the context variable of the DI Job.

   Select *Simple* from the **Language** list.

   In the **Expression** field, enter *${header.camelfilename}* to get the file name.

7. Double-click **cTalendJob** to open its **Basic settings** view in the **Component** tab.



Call_DI-Job(cTalendJob_1)

Library: cTalendJob_ShowContextVar-0.1.jar

Job: "work.ctalendjob_showcontextvar_0_1.cTalendJob_ShowContextVar"

Context: "Default"

8. Click the **[...]** button to browse the generated jar file for the DI Job.

9. Go to the unzipped folder of the above JAR file and open the *job.xml* in the *<DI_Job_JAR_Path>\OSGI-INF\blueprint* folder, *E:\cTalendJob_ShowContextVar-0.1\OSGI-INF\blueprint* in this example.

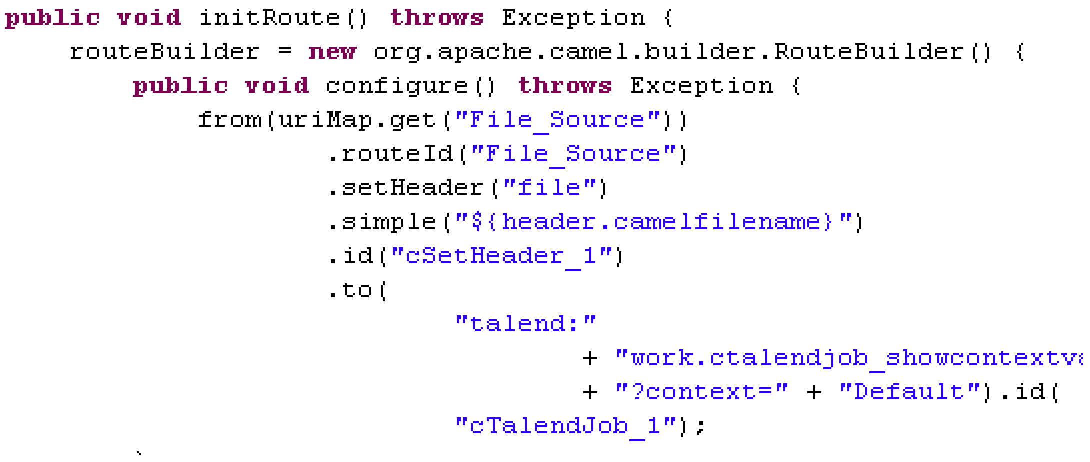
   Go to the *bean* tag and copy the content of the attribute *class*, *work.ctalendjob_showcontextvar_0_1.cTalendJob_ShowContextVar* in this example.

   Paste it in the **Job** field.

10. Press **Ctrl+S** to save the Route.

Viewing the code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.



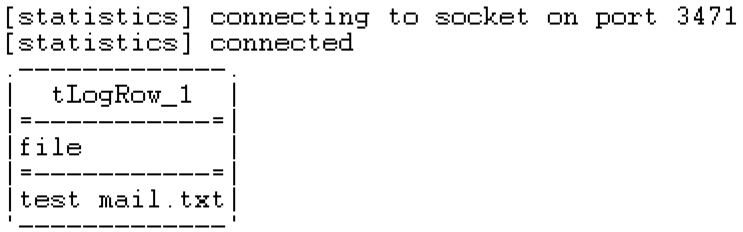
```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("File_Source"))
                    .routeId("File_Source")
                    .setHeader("file")
                    .simple("${header.camelfilename}")
                    .id("cSetHeader_1")
                    .to(
                            "talend:"
                                    + "work.ctalendjob_showcontextv
                                    + "?context=" + "Default").id(
                            "cTalendJob_1");
```

   As shown above, `File_Source` provides a file for the message exchange, `cSetHeader` sets a message header and uses the source file name as the header value, and finally that value is passed to `cTalendJob_1` for execution of the DI Job.

2. Press **F6** to execute the Route.

3. Put a file into the folder specified by *context.root_dir*, *test mail.txt* in this example.

   The result below can be found.



```
[statistics] connecting to socket on port 3471
[statistics] connected
.-------------.
|  tLogRow_1  |
|=-----------=|
|file         |
|=-----------=|
|test mail.txt|
'-------------'
```

As shown above, the source file name is displayed via **tLogRow** as the Route's message header value has been passed to the context variable of the DI Job.

Miscellaneous components

This chapter details the major components that you can find in **Miscellaneous** family from the **Palette** of the **Mediation** perspective of *Talend Open Studio for ESB*.

The **Miscellaneous** family groups components that cover the needs such as iterating a Route or stopping a Route.

cLog

cLog properties

Component Family	Miscellaneous	
Function	**cLog** logs message exchanges to the underlying logging mechanism. Apache Camel provides the regular logger and the throughput logger. The default logger logs every exchange. The throughput logger logs exchanges on a group basis. By default regular logging is used.	
Purpose	**cLog** is used to log message exchanges.	
	Level	Select a logging level from **DEBUG**, **ERROR**, **INFO**, **OFF**, **TRACE**, or **WARN**.
	Use default output log message	Select this option to use the default output log message provided by the underlying logging mechanism.
	Options / None (For default output log message only)	Select this option to take no action on the log message.
	Options / Specifies a group size for throughput logging (For default output log message only)	Select this option to use throughput logging and specify a group size for the throughput logging. **Size**: Enter an integer that specifies a group size for throughput logging.
	Options / Group message stats by time interval (in millis) (For default output log message only)	Select this option to use throughput logging and group message statistics. **Interval**: Specify the time interval (in milliseconds) by which the message statistics will be grouped. **Delay**: Set the initial delay (in milliseconds) for message statistics.
	Options / Format the log output (For default output log message only)	Select this option to format the log output. Click **[+]** as many times as required to add arguments to the table. Then click the corresponding **value** field and enter a value. See the site http://camel.apache.org/log.html for available options.
	Specify output log message	Select this option to specify the output log message. **Message**: Use Simple language to construct a dynamic message which gets logged.
Usage	**cLog** is used as a middle or end component in a Route.	
Limitation	n/a	

Related scenario:

For a related scenario, see the section called “Scenario: Routing messages according to a criterion”.

cLoop

cLoop properties

Component Family	Miscellaneous	
Function	**cLoop** allows you to process a message or messages a number of times and possibly in different ways.	
Purpose	**cLoop** is used to process a message or messages repetitively.	
Basic settings	*Loop Type*	Select a type of loop to be carried out: **Expression**, **Header**, or **Value**. **Expression**: Use an expression to determine the loop count. **Header**: Use a header to determine the loop count. **Value**: Use an argument to set the loop count.
		When using **Expression**: In the **Language** field, select the language of the expression you want to use to determine the loop count between **Constant**, **EL**, **Groovy**, **Header**, **Javascript**, **JoSQL**, **JXPath**, **MVEL**, **None**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, **XQuery**. Type in the expression in the **Expression** field.
		When using **Header**: Enter the name of the header that you want to use to determine the loop count in **header** field.
		When using **Value**: Enter an integer you want to set as the loop count in the **value** field.
Usage	**cLoop** can be a middle component in a Route.	
Limitation	n/a	

Related scenario:

No scenario is available for this component yet.

cStop

cStop properties

Component Family	Miscellaneous
Function	**cStop** stops the Route to which it is connected.
Purpose	**cStop** stops the Route to which it is connected.
Usage	**cStop** is not a start component, but it can be a middle or end component in a Route.
Limitation	n/a

Related scenario:

For a related scenario, see the section called “Scenario: Intercepting several routes and redirect them in a single new route” of the section called “cIntercept”.

Processor components

This chapter details the major components that you can find in **Processor** family from the **Palette** of the **Mediation** perspective of *Talend Open Studio for ESB*.

The **Processor** family groups components that help you to perform all types of processing tasks on message flows such monitoring the message sent or received, setting the message exchange mode, controlling the delivery time, and so on.

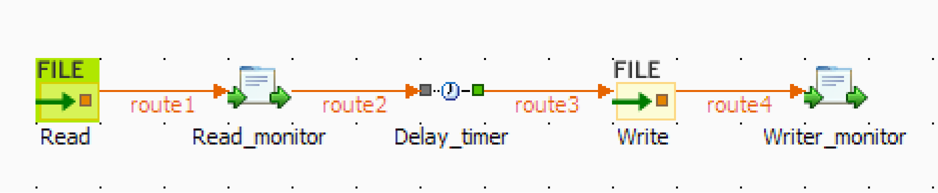
cDelayer

cDelayer properties

Component Family	Processor	
Function	The **cDelayer** component delays the delivery of messages.	
Purpose	The **cDelayer** component allows you to set a latency in message routing.	
Basic settings	*Time to wait (in ms)*	Fill this field with an integer (in milliseconds) to define the time to wait before sending the message to the subsequent endpoint.
Usage	This component is usually used in the middle of a Route.	
Limitation	n/a	

Scenario: Using cDelayer to delay message routing

In this scenario, a **cDelayer** component is used to delay the routing of each message to the target endpoint by 20 seconds.



Dropping and linking the components

This use case requires one **cDelayer** component, two **cFile** components, and two **cProcessor** components.

1. From the **Messaging** folder of the **Palette**, drop two **cFile** components onto the design workspace, one to read files from a local folder and the other to write the files to another local folder.

2. From the **Processor** folder of the **Palette**, drop two **cProcessor** components onto the design workspace, one next to the reading component to monitor messages read from the source file folder, and the other next to the writing component to monitor messages written to the target file folder.

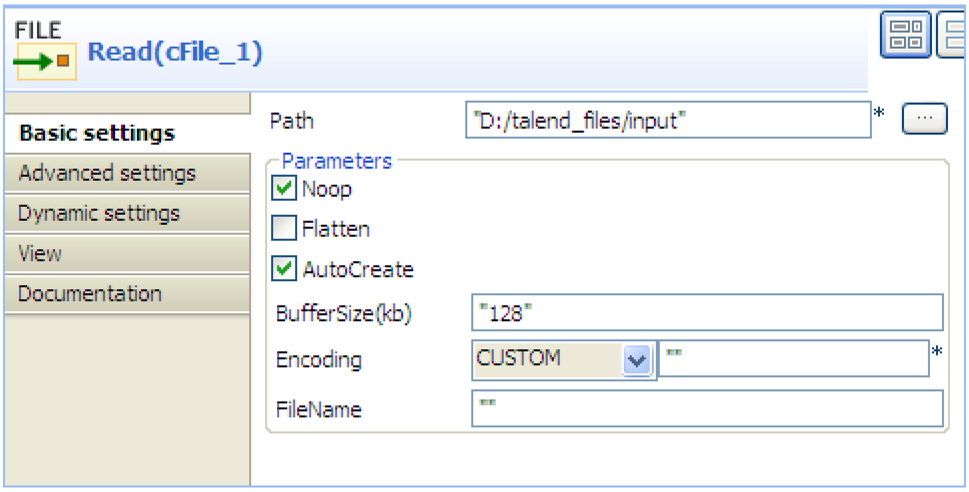
3. From the **Processor** folder of the **Palette**, drop one **cDelayer** component onto the design workspace, between the message reading monitor component and the message writing component.

4. Connect the components using **Row** > **Route** connections.

5. Label the components to better identify their roles in the Route, as shown above.

Configuring the components

1. Double-click the first **cFile** component, which is labelled *Read*, to open its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/input"

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

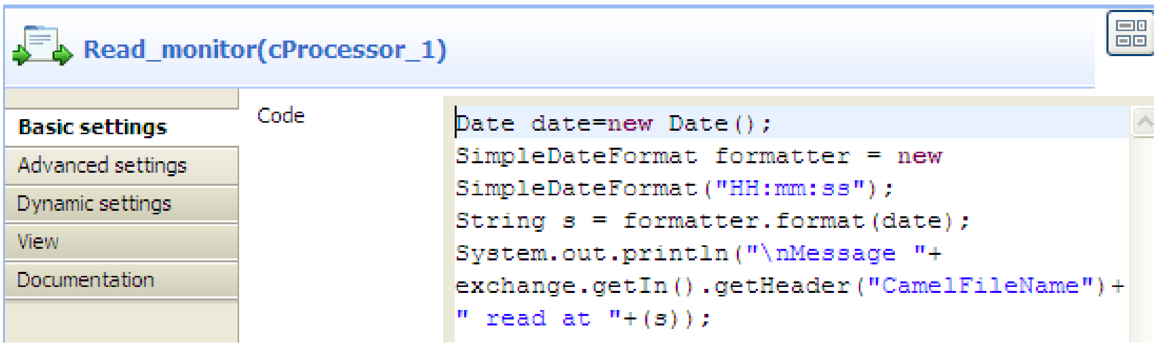
Encoding CUSTOM ""

FileName ""

2. In the **Path** field, enter or browse to the path to the source files, and leave the other parameters as they are.

3. Repeat these steps to define the target folder in property settings of the second **cFile** component, which is labelled *Write*.

4. Double-click the first **cProcessor** component, which is labelled *Read_monitor*, to open its **Basics settings** view in the **Component** tab.



Code

```
Date date=new Date();
SimpleDateFormat formatter = new
SimpleDateFormat("HH:mm:ss");
String s = formatter.format(date);
System.out.println("\nMessage "+
exchange.getIn().getHeader("CamelFileName")+
" read at "+(s));
```

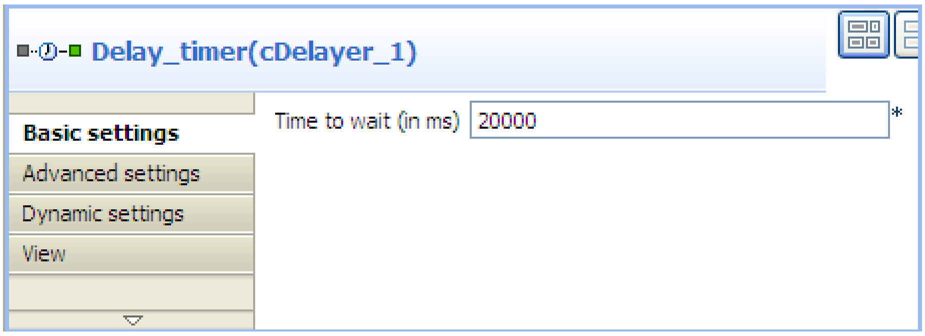
5. In the **Code** area, customize the code to display the time each message is read from the source:

```
Date date=new Date();
SimpleDateFormat formatter = new SimpleDateFormat("HH:mm:ss");
String s = formatter.format(date);
System.out.println("\nMessage "+
exchange.getIn().getHeader("CamelFileName")+
" read at "+(s));
```

6. Repeat these steps to configure the second **cProcessor** component, which is labelled *Write_monitor*, to display the time each message is written to the target:

```
Date date=new Date();
SimpleDateFormat formatter = new SimpleDateFormat("HH:mm:ss");
String s = formatter.format(date);
System.out.println("Message "+
exchange.getIn().getHeader("CamelFileName")+ " written at "+(s));
```

7. Double-click the **cDelayer** component, which is labelled *Delay_timer*, to open its **Basic settings** view in the **Component** tab.



Delay_timer(cDelayer_1)

Time to wait (in ms) 20000 *

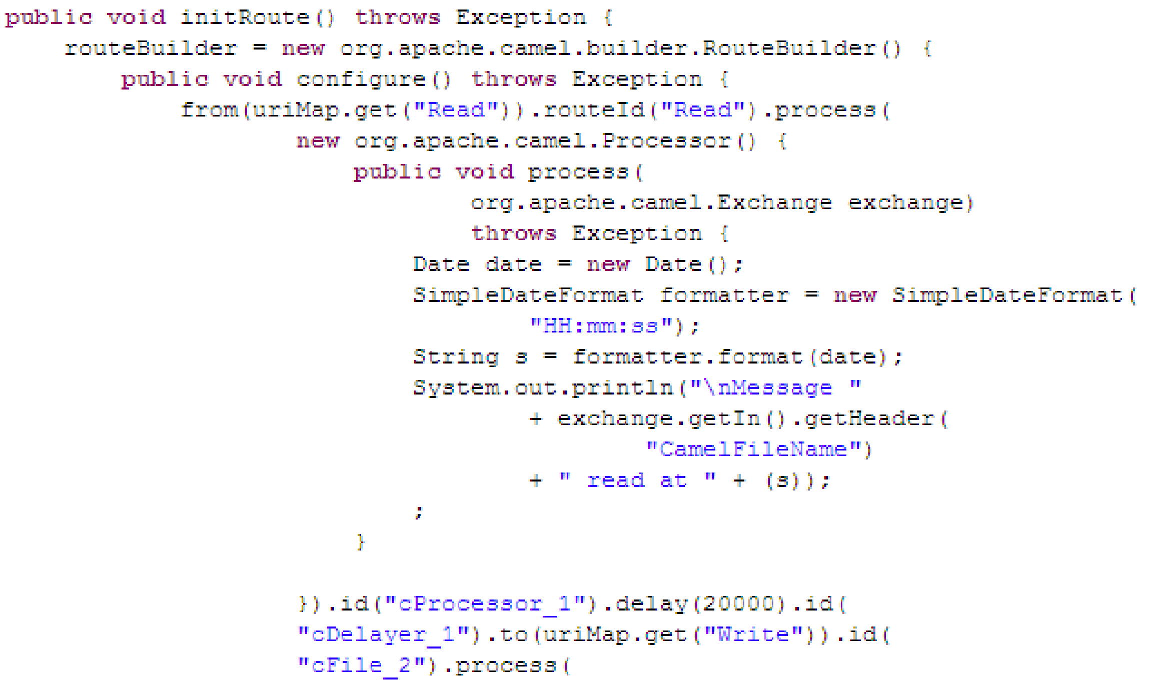
8. In the **Time to wait (in ms)** field, enter the number of milliseconds by which you want to delay message delivery. Note that the value must be a positive integer.

   In this use case, we want each message to be delivered after a 20-second delay.

9. Press **Ctrl+S** to save your Route.

Viewing the code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

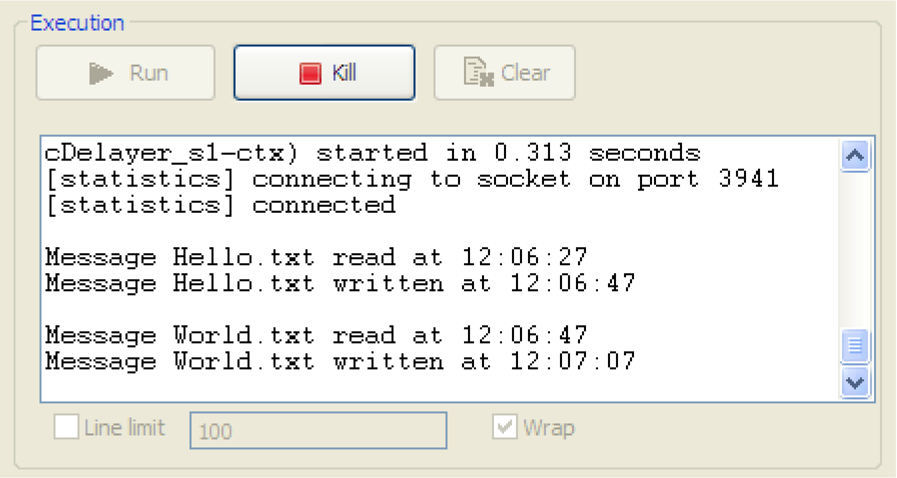


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Read")).routeId("Read").process(
                    new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            Date date = new Date();
                            SimpleDateFormat formatter = new SimpleDateFormat(
                                    "HH:mm:ss");
                            String s = formatter.format(date);
                            System.out.println("\nMessage "
                                    + exchange.getIn().getHeader(
                                            "CamelFileName")
                                    + " read at " + (s));
                            ;
                        }

                    }).id("cProcessor_1").delay(20000).id(
                    "cDelayer_1").to(uriMap.get("Write")).id(
                    "cFile_2").process(
```

   As shown in the code, a 20-second delay is implemented according to `.delay(20000)` in the message routing `from` the `Read` endpoint `.to` the `Write` endpoint.

2. Press **F6** to execute the Route.

   RESULT: Each message read from the source folder is routed to the target folder after a 20-second delay.



```
cDelayer_s1-ctx) started in 0.313 seconds
[statistics] connecting to socket on port 3941
[statistics] connected

Message Hello.txt read at 12:06:27
Message Hello.txt written at 12:06:47

Message World.txt read at 12:06:47
Message World.txt written at 12:07:07
```

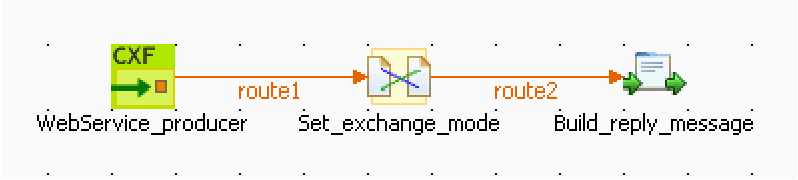
cExchangePattern

cExchangePattern properties

Component Family	Processor	
Function	**cExchangePattern** can be configured to indicate the message exchange mode.	
Purpose	**cExchangePattern** allows you to set the message exchange mode.	
Basic settings	*Exchange Patterns*	Select the message exchange mode from **InOnly** or **InOptionalOut**, **InOut**, **OutIn**, **OutOptionalIn**, **RobustInOnly**, **RobustOutOnly**.
Usage	As a middle component in a Route, **cExchangePattern** allows you to set the message exchange mode.	
Limitation		

Scenario: Enabling the InOut exchange pattern to get replies

In this scenario, a **cExchangePattern** component is used to enable the request/reply exchange pattern in the Route, so that the client can get a reply from the server.



To send requests to the server side, a soapUI is needed and its configuration will be briefed in the following contents.

To build the Route, do the following.

Dropping and linking the components

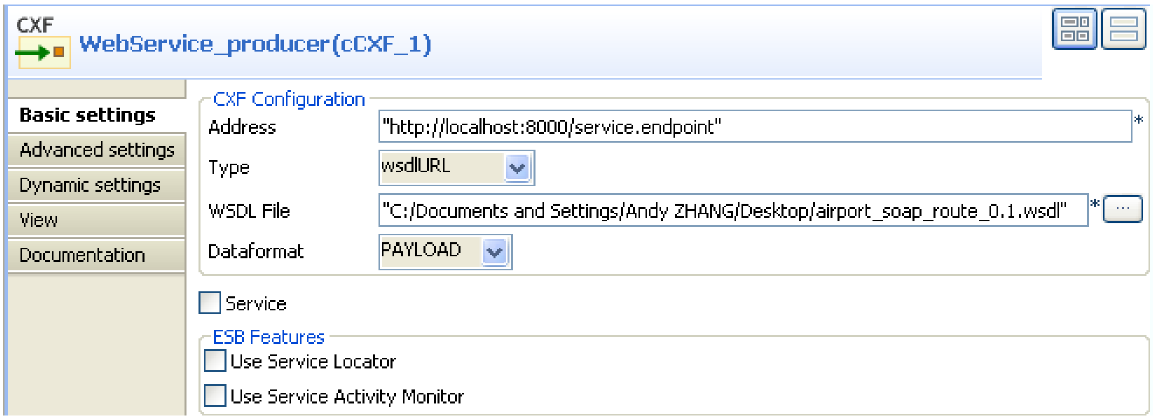
1. From the **Processor** folder of the **Palette**, drag and drop a **cCXF**, a **cExchangePattern** and a **cProcessor** onto the workspace, and label them *WebService_producer*, *Set_exchange_mode* and *Build_reply_message* respectively to better identify their roles in the Route.

2. Link **cCXF** to **cExchangePattern** using a **Row** > **Route** connection.

3. Link **cExchangePattern** to **cProcessor** using a **Row** > **Route** connection.

Configuring the components

1. Double-click **cCXF** to open its **Basic settings** view in the **Component** tab.



CXF Configuration

Address "http://localhost:8000/service.endpoint" *

Type wsdlURL

WSDL File "C:/Documents and Settings/Andy ZHANG/Desktop/airport_soap_route_0.1.wsdl" *

Dataformat PAYLOAD

□ Service

ESB Features

□ Use Service Locator

□ Use Service Activity Monitor

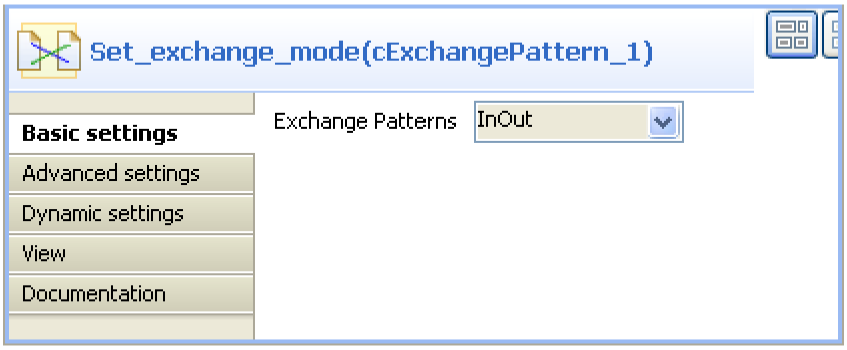
2. In the **Address** field, leave the default setting unchanged.

3. In the **Type** list, select **wsdlURL**.

4. In the **WSDL File** field, enter the URL of the wsdl file. You can also click the three-dot button to browse for it.

5. In the **Dataformat** list, select **PAYLOAD**.

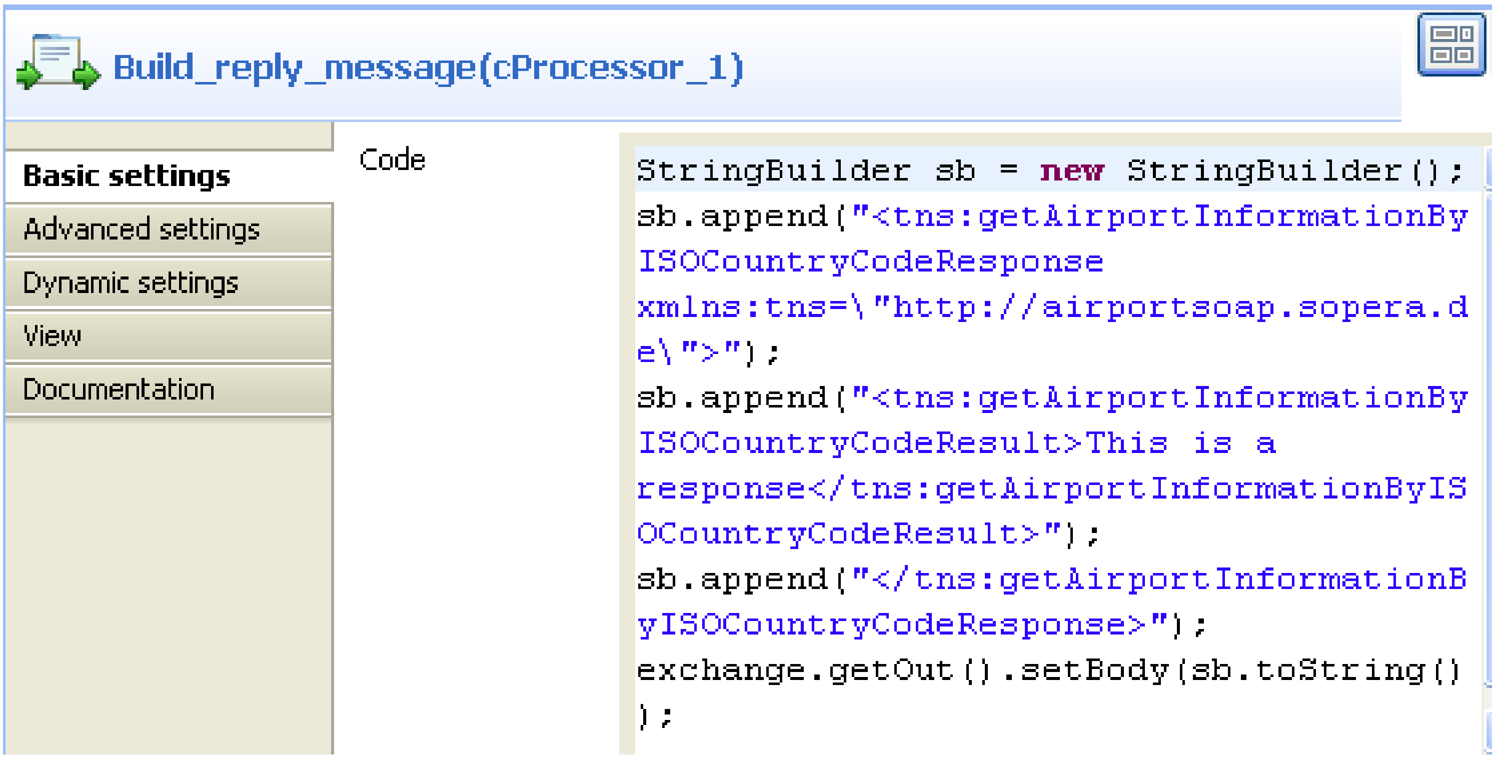
6. Double-click **cExchangePattern** to open its **Basic settings** view in the **Component** tab.



Exchange Patterns InOut

7. In the **Exchange Patterns** list, select **InOut** to enable the request/reply message exchange mode.

8. Double-click **cProcessor** to open its **Basic settings** view in the **Component** tab.



```
StringBuilder sb = new StringBuilder();
sb.append("<tns:getAirportInformationBy
ISOCountryCodeResponse
xmlns:tns=\"http://airportsoap.sopera.d
e\">");
sb.append("<tns:getAirportInformationBy
ISOCountryCodeResult>This is a
response</tns:getAirportInformationByIS
OCountryCodeResult>");
sb.append("</tns:getAirportInformationB
yISOCountryCodeResponse>");
exchange.getOut().setBody(sb.toString()
);
```

9. In the **Code** box, enter the code below.

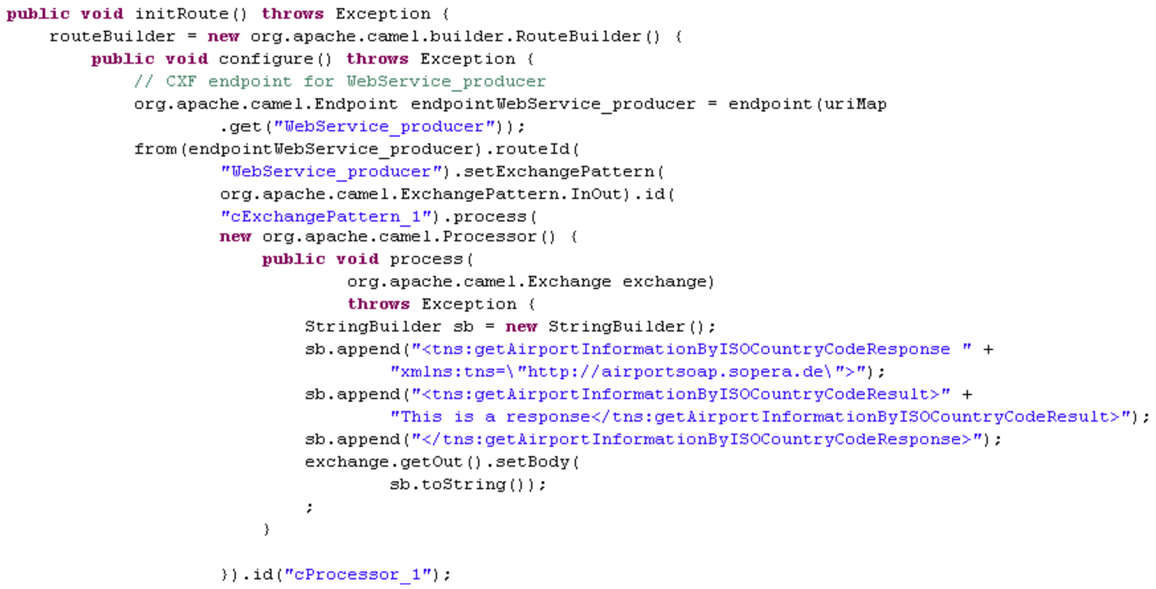
```
StringBuilder sb = new StringBuilder();
sb.append("<tns:getAirportInformationByISOCountryCodeResponse
 xmlns:tns=\"http://airportsoap.sopera.de\">");
sb.append("<tns:getAirportInformationByISOCountryCodeResult>This is a
 response</tns:getAirportInformationByISOCountryCodeResult>");
sb.append("</tns:getAirportInformationByISOCountryCodeResponse>");
exchange.getOut().setBody(sb.toString());
```

As shown above, a string is built here and is used as a reply message of the route. It is in line with the message definition of the above wsdl file.

10. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.



```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            // CXF endpoint for WebService_producer
            org.apache.camel.Endpoint endpointWebService_producer = endpoint(uriMap
                    .get("WebService_producer"));
            from(endpointWebService_producer).routeId(
                    "WebService_producer").setExchangePattern(
                    org.apache.camel.ExchangePattern.InOut).id(
                    "cExchangePattern_1").process(
                    new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            StringBuilder sb = new StringBuilder();
                            sb.append("<tns:getAirportInformationByISOCountryCodeResponse " +
                                    "xmlns:tns=\"http://airportsoap.sopera.de\">");
                            sb.append("<tns:getAirportInformationByISOCountryCodeResult>" +
                                    "This is a response</tns:getAirportInformationByISOCountryCodeResult>");
                            sb.append("</tns:getAirportInformationByISOCountryCodeResponse>");
                            exchange.getOut().setBody(
                                    sb.toString());
                            ;
                        }

                    }).id("cProcessor_1");
```

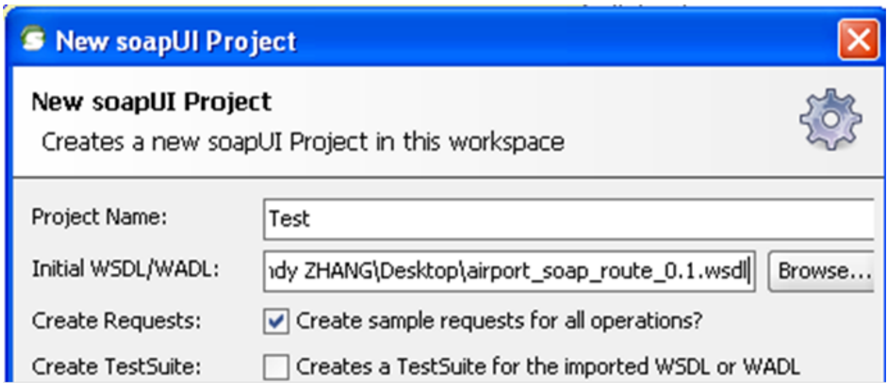
As shown above, the route has its message exchange pattern set as `InOut` using the method `.setExchangePattern(org.apache.camel.ExchangePattern.InOut)`. In the meantime, a string is created using `StringBuilder sb = new StringBuilder()` at `cProcessor_1` and is used as the reply message via the method `exchange.getOut().setBody( sb.toString())`.

2. Press **F6** to execute the Route.

   The server Route gets started.

Creating and sending a request to the server Route and getting a reply

1. In the soapUI, create a Test project and edit a request, as illustrated below:



New soapUI Project

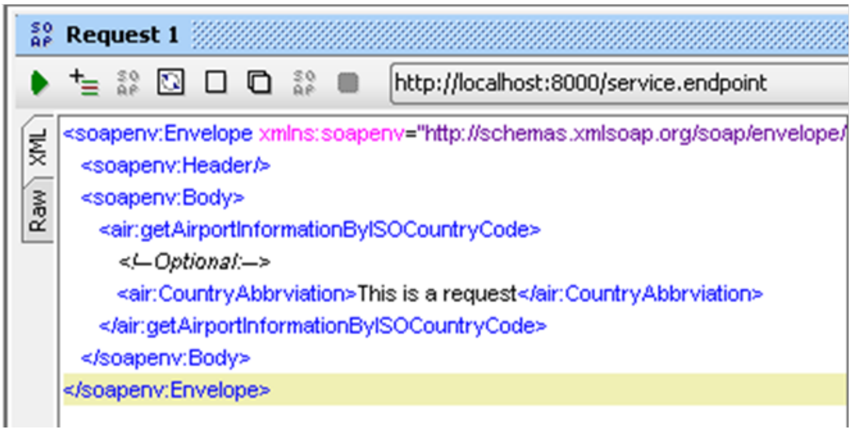
Creates a new soapUI Project in this workspace

Project Name: Test

Initial WSDL/WADL: ıdy ZHANG\Desktop\airport_soap_route_0.1.wsdl

Create Requests: ☑ Create sample requests for all operations?

Create TestSuite: □ Creates a TestSuite for the imported WSDL or WADL



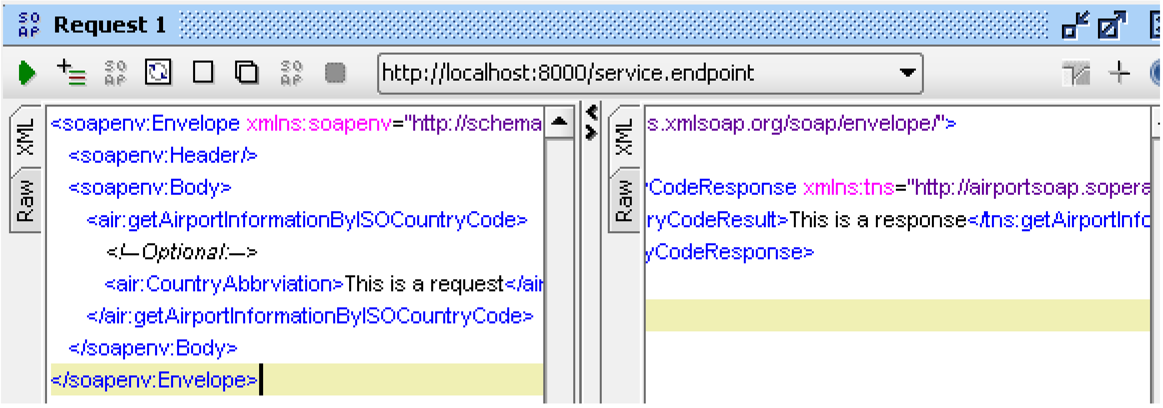
Request 1

http://localhost:8000/service.endpoint

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/
  <soapenv:Header/>
  <soapenv:Body>
    <air:getAirportInformationByISOCountryCode>
      <!--Optional:-->
      <air:CountryAbbrviation>This is a request</air:CountryAbbrviation>
    </air:getAirportInformationByISOCountryCode>
  </soapenv:Body>
</soapenv:Envelope>
```

Note that the wsdl file must be same as that configured for **cCXF**, so that the request can be in line with the definition of the web service.

2. Send the request to the server Route and you can get the reply, as illustrated below:



Request 1

http://localhost:8000/service.endpoint

```
<soapenv:Envelope xmlns:soapenv="http://schema
  <soapenv:Header/>
  <soapenv:Body>
    <air:getAirportInformationByISOCountryCode>
      <!--Optional:-->
      <air:CountryAbbrviation>This is a request</air
    </air:getAirportInformationByISOCountryCode>
  </soapenv:Body>
</soapenv:Envelope>
```

```
s.xmlsoap.org/soap/envelope/">

CodeResponse xmlns:tns="http://airportsoap.sopera
ryCodeResult>This is a response</tns:getAirportInfo
yCodeResponse>
```

cJavaDSLProcessor

cJavaDSLProcessor properties

Component Family	Processor	
Function	**cJavaDSLProcessor** implements producers and consumers of message exchanges or implements a Message Translator using the Java Domain Specific Language (DSL).	
Purpose	**cJavaDSLProcessor** can be usable for quickly whirling up some code using Java DSL. If the code in the inner class gets a bit more complicated it is of course advised to refactor it into a separate class.	
Basic settings	*Code*	Type in the code you want to implement using Java DSL.
Usage	**cJavaDSLProcessor** is used as a middle or end component in a Route.	
Limitation	n/a	

Related scenario:

For a related scenario, see the section called “Scenario: Wiretapping a message in a Route”.

cProcessor

cProcessor properties

Component Family	Processor	
Function	**cProcessor** implements consumers of message exchanges or implements a Message Translator.	
Purpose	**cProcessor** can be usable for quickly whirling up some code. If the code in the inner class gets a bit more complicated it is of course advised to refactor it into a separate class.	
Basic settings	*Code*	Type in the Java code you want to implement.
Usage	**cProcessor** is used as a middle or end component in a Route.	
Limitation	n/a	

Related scenario:

For a related scenario, see the section called “Scenario: Intercepting several routes and redirect them in a single new route” of the section called “cIntercept”.

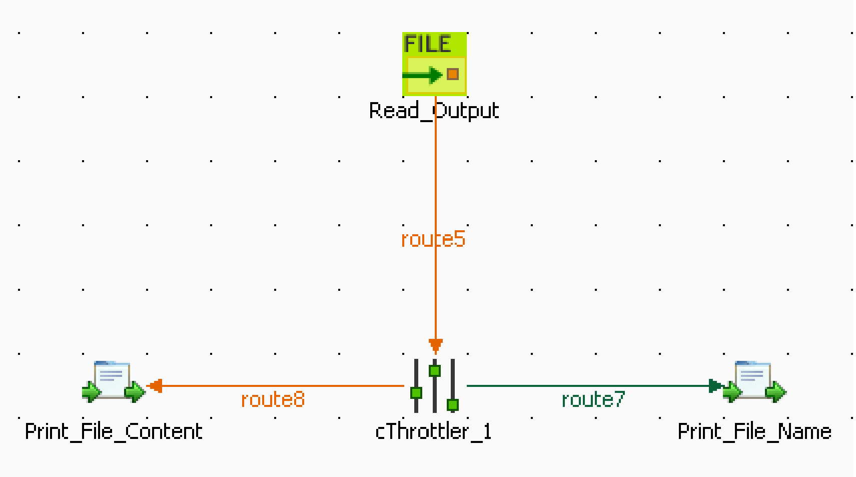
cThrottler

cThrottler properties

Component Family	Processor	
Function	**cThrottler** is designed to limit the number of messages flowing to the subsequent endpoint.	
Purpose	**cThrottler** allows you to limit the number of messages flowing to a specific endpoint in order to prevent it from getting overloaded.	
Basic settings	*Request per period*	The number of messages allowed to pass **cThrottler** within the defined time period.
	Set time period	Select this check box to set the value of the time period (in milliseconds) and enable throttling.
	Use asynchronous delaying	If this check box is selected, any messages that are delayed will be routed asynchronously using a scheduled thread pool.
Usage	Being a middle component, **cThrottler** allows you to limit the number of messages flowing to a specific endpoint in order to prevent it from getting overloaded.	
Connections	*throttler*	Select this link to route the throttled messages to the next endpoint.
	Route	Select this link to route all the messages from the sender to the next endpoint.
Limitation	n/a	

Scenario: Throttling the message flow

In this scenario, a **cThrottler** component is used to reduce the number of messages flowing out within a time period.



To build the Route, do the following.

Dropping and linking the components

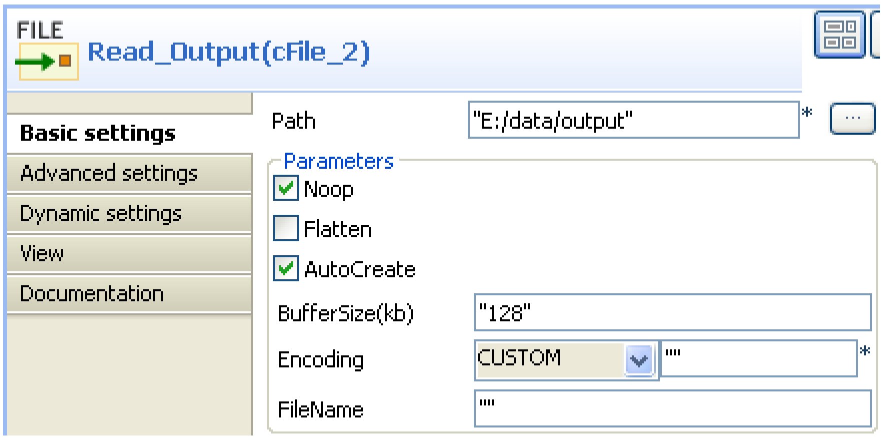
1. Drag and drop the components from the **Palette** onto the workspace: **cThrottler**, **cFile** and two **cProcessor**. Change the label of the **cFile** component to **Read_Output**. Change the labels of the two **cProcessor** components to **Print_File_Name** and **Print_File_Content**.

2. Link **Read_Output** to **cThrottler** using a **Row** > **Route** connection.

3. Link **cThrottler** to **Print_File_Name** using a **Row** > **Throttler** connection, and to **Print_File_Content** using a **Row** > **Route** connection.

Configuring the components

1. Double-click **Read_Output** to open its **Basic settings** view in the **Component** tab.



Read_Output(cFile_2)

Path "E:/data/output"

Parameters

☑ Noop

☐ Flatten

☑ AutoCreate

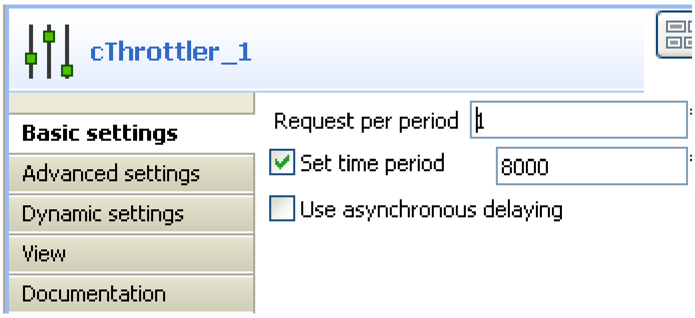
BufferSize(kb) "128"

Encoding CUSTOM ""

FileName ""

2. In the **Path** field, type in the path to the source message, for example, *"E:/data/output"*. Keep the default settings for other fields.

3. Double-click **cThrottler** to open its **Basic settings** view in the **Component** tab.



cThrottler_1

Request per period 1

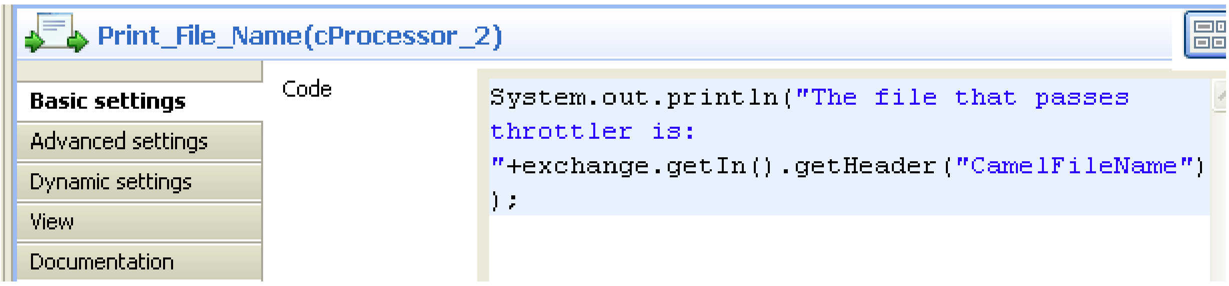
☑ Set time period 8000

☐ Use asynchronous delaying

4. In the **Request per period** field, type in the number of messages allowed to pass the throttler per period, for example, *1*.

   In the **Set time period** field, type in the value of the period, for example, *8000*.

5. Double-click **Print_File_Name** to open its **Basic settings** view in the **Component** tab.

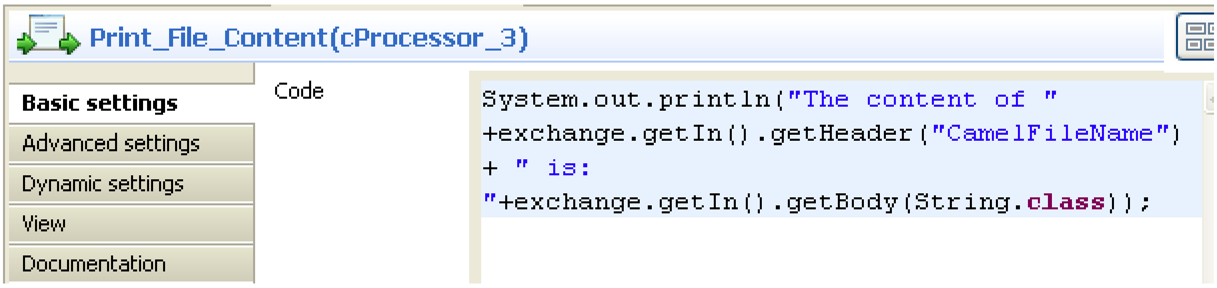


```
System.out.println("The file that passes
throttler is:
"+exchange.getIn().getHeader("CamelFileName")
);
```

6. In the **Code** box, enter the code below to get the name of the message that passes the throttler.

```
System.out.println("The file that passes throttler is:
 "+exchange.getIn().getHeader("CamelFileName"));
```

7. Double-click **Print_File_Content** to open its **Basic settings** view in the **Component** tab.



```
System.out.println("The content of "
+exchange.getIn().getHeader("CamelFileName")
+ " is:
"+exchange.getIn().getBody(String.class));
```

8. In the **Code** box, enter the code below to get the content of the message that passes the throttler.

```
System.out.println("The content of "
 +exchange.getIn().getHeader("CamelFileName")+ " is: "
+exchange.getIn().getBody(String.class));
```

9. Press **Ctrl+S** to save your Route.

Viewing the code and executing the Route

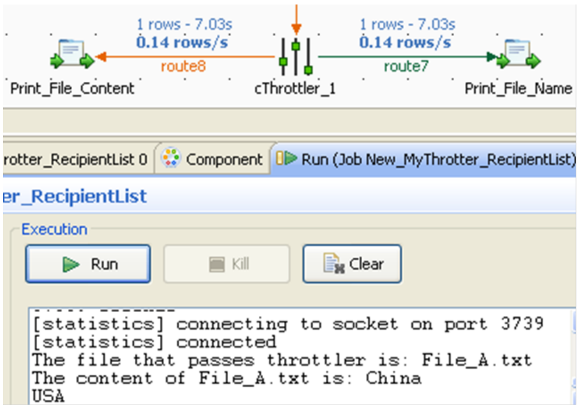
1. Click the **Code** tab at the bottom of the design workspace to check the generated code.

```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Read_Output")).routeId("Read_Output")
                    .throttle(1).timePeriodMillis(8000).id(
                            "cThrottler_1").process(
                            new org.apache.camel.Processor() {
                                public void process(
                                        org.apache.camel.Exchange exchange)
                                        throws Exception {
                                    System.out
                                            .println("The file that passes throttler is: "
                                                    + exchange
                                                            .getIn()
                                                            .getHeader(
                                                                    "CamelFileName"));
                                    ;
                                }

                            }).id("cProcessor_2").end().process(
                            new org.apache.camel.Processor() {
                                public void process(
                                        org.apache.camel.Exchange exchange)
                                        throws Exception {
                                    System.out
                                            .println("The content of "
                                                    + exchange
                                                            .getIn()
                                                            .getHeader(
                                                                    "CamelFileName")
                                                    + " is: "
                                                    + exchange
                                                            .getIn()
                                                            .getBody(
                                                                    String.class));
                                    ;
```

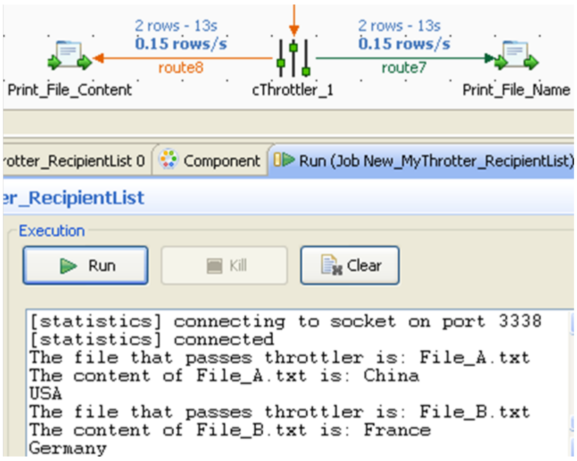
As shown above, the messages from `Read_Output` go through throttling at `cThrottler_1`, with only `(1)` message allowed to leave the throttler within each `timePeriodMillis(8000)`. Meanwhile, the filename and the content of the throttled message are printed out via the two processors.

2. Press **F6** to execute the Route.

   As shown below, *File_A.txt* was delivered within the first time period while in the second period, *File_B.txt* was delivered as well.



```
[statistics] connecting to socket on port 3739
[statistics] connected
The file that passes throttler is: File_A.txt
The content of File_A.txt is: China
USA
```



```
[statistics] connecting to socket on port 3338
[statistics] connected
The file that passes throttler is: File_A.txt
The content of File_A.txt is: China
USA
The file that passes throttler is: File_B.txt
The content of File_B.txt is: France
Germany
```

Routing components

This chapter details the major components that you can find in **Routing** family from the **Palette** of the **Mediation** perspective of *Talend Open Studio for ESB*.

The **Routing** family groups components that moves messages from one endpoint to another based on a set of conditions.

cAggregate

cAggregate

Component Family	Routing	
Function	**cAggregate** aggregates messages together according to specified conditions.	
Purpose	**cAggregate** allows you to combine a number of messages together into a single message.	
Basic settings	*Language*	Select the language of the expression you want to use to filter your messages, from **Constant**, **EL**, **Groovy**, **Header**, **Javascript**, **JoSQL**, **JXPath**, **MVEL**, **None**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**.
	Correlation expression/ Expression	Type in the expression that evaluates the correlation key to be used for the aggregation.
	Strategy	Specify a Java bean to use as the aggregation strategy.
	Completion conditions/ Number of messages	Select this check box to specify the number of messages to aggregate per batch before the aggregation is complete. 💡 By default, this check box is selected and the number of messages is set to *3*. If you clear this check box, and at least one of the other four completion conditions is met, all the messages retrieved will be aggregated in one batch.
	Completion conditions/ Inactivity timeout (in milliseconds)	Select this check box to specify the time (in milliseconds) that an aggregated exchange should be inactive before it is complete. This option can be set as either a fixed value or using an Expression which allows you to evaluate a timeout dynamically. 💡 You can not use this option together with **Scheduled interval**. Only one of them can be used at a time.
	Completion conditions/ Scheduled interval (in milliseconds)	Select this check box to specify a repeating period (in milliseconds) by which the aggregator will complete all current aggregated exchanges. 💡 You cannot use this option together with **Inactivity timeout**. Only one of them can be used at a time.
	Completion conditions/ Predicate matched	Select this check box to specify a predicate to indicate when an aggregated exchange is complete.
	Completion conditions/ Batch consumer	Select this check box to aggregate all files consumed from a file endpoint in a given poll.
Advanced settings	*Check completion before aggregating*	Select this check box to check for completion when a new incoming exchange has been received. This option

		influences the behavior of the **Predicate matched** option as the exchange being passed in changes accordingly. When this option is disabled, the exchange passed in the predicate is the *aggregated* exchange which means any information you may store on the aggregated exchange from the aggregation strategy is available for the predicate. When this option is enabled, the exchange passed in the predicate is the *incoming* exchange, which means you can access data from the incoming exchange.
	Close correlation group	Select this check box to indicate that if a correlation key has already been completed, then any new exchanges with the same correlation key will be denied. When using this option, enter a number in the **Maximum bound** field to keep that last number of closed correlation keys.
	Ignore invalid correlation key	Select this check box to ignore the invalid correlation key which could not be evaluated to a value. By default Camel will throw an Exception on encountering an invalid correlation key.
	Group arriving exchange	Select this check box to group all aggregated exchanges into a single combined holder class that holds all the aggregated exchanges. As a result only one exchange is being sent out from the aggregator. This option can be used to combine many incoming exchanges into a single output exchange.
	Use persistence	Select this check box to plug in your own implementation of the repository which keeps track of the current in-flight aggregated exchanges. By default, Camel uses a memory based implementation.
	Repository	This field appears when the **Use persistence** check box is selected. The repository is **AggregationRepository**, **HawtDBAggregationRepository**, or **RecoverableAggregationRepository**.
		AggregationRepository: The default repository used by Camel which is a memory based implementation. Enter the name of the repository in the field.
		HawtDBAggregationRepository: HawtDBAggregationRepository is an AggregationRepository which persists the aggregated messages on the fly. This ensures that you will not loose messages. With this repository selected, the following options appear: **Use persistent file**: Select this check box to store the aggregated exchanges in a file. Enter the name of the file for the persistent storage in the **Persistent file** field. If the file does not exists on startup, it will be created. **Recovery/Use recovery**: Select this check box to recover failed aggregated exchanges and have them resubmitted automatically. In the **Recovery interval** field, enter the interval (in milliseconds) to scan for failed exchanges to recover and resubmit. By default this interval is 5000 milliseconds. In the **Dead letter channel** field, enter an endpoint URI for a Dead Letter Channel where exhausted recovered exchanges will be moved.

		In the **Maximum redeliveries** field, enter the maximum number of redelivery attempts for a given recovered exchange.
		RecoverableAggregationRepository: RecoverableAggregationRepository is a JDBC based AggregationRepository which persists the aggregated messages on the fly. This ensures that you will not loose messages. Enter the name of the repository in the field. With this repository selected, the following options appear: **Recovery/Use recovery**: Select this check box to recover failed aggregated exchanges and have them resubmitted automatically. In the **Recovery interval** field, enter the interval (in milliseconds) to scan for failed exchanges to recover and resubmit. By default this interval is 5000 milliseconds. In the **Dead letter channel** field, enter an endpoint URI for a Dead Letter Channel where exhausted recovered exchanges will be moved. In the **Maximum redeliveries** field, enter the maximum number of redelivery attempts for a given recovered exchange.
Usage	**cAggregate** is used as a middle or end component in a Route.	
Connections	*Aggregate*	Select this link to route messages to the next endpoint according to the selected aggregation strategy.
	Route	Select this link to route all the messages from the sender to the next endpoint.
Limitation	n/a	

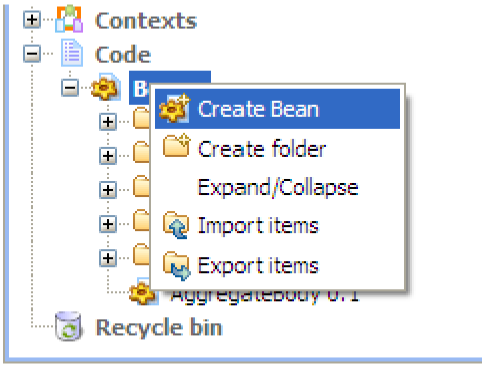
Scenario: Aggregating three messages into one

In this scenario, the **cAggregate** component combines three messages from the local file system into one and prints the messages in the console. A Java bean will be used as the aggregation strategy.

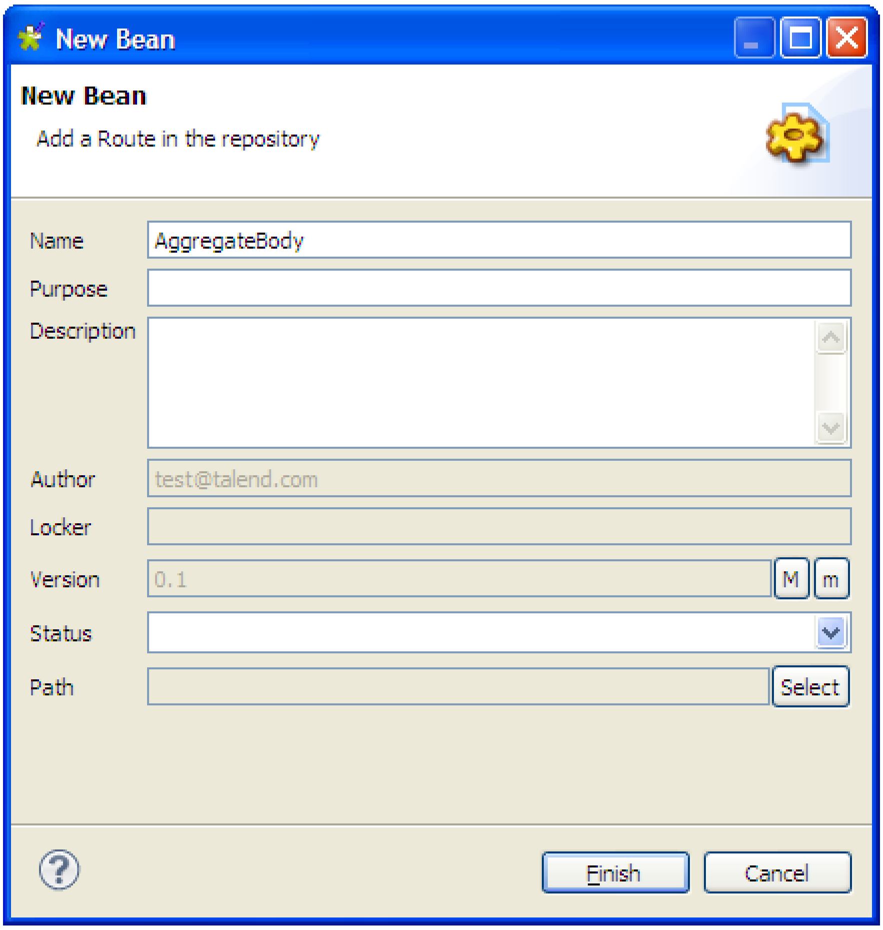
Creating a Java bean as the aggregation strategy

To aggregate the messages, we will use a Java bean that will help us build an aggregation strategy.

1. From the repository tree view, expand the **Code** node and right click the **Beans** node. In the contextual menu, select **Create Bean**.



2. The **New Bean** wizard opens. In the **Name** field, type in a name for the bean, for example, *AggregateBody*. Click **Finish** to close the wizard.



New Bean

New Bean

Add a Route in the repository

Name: AggregateBody

Purpose:

Description:

Author: test@talend.com

Locker:

Version: 0.1

Status:

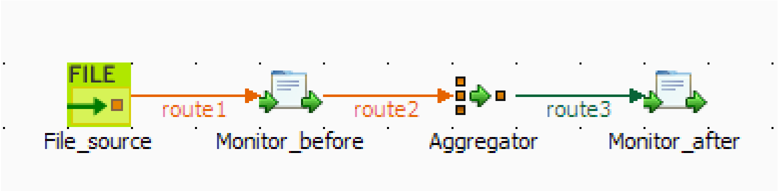
Path:

3. Type in the codes as shown in the figure below. In this use case, we just want to aggregate all messages into a single message.

```
package beans;

import org.apache.camel.Exchange;
import org.apache.camel.processor.aggregate.AggregationStrategy;

public class AggregateBody implements AggregationStrategy{

 public Exchange aggregate(Exchange oldEx, Exchange newEx) {
  if(oldEx==null){
   return newEx;
  }
  String oldBody = oldEx.getIn().getBody(String.class);
  String newBody = newEx.getIn().getBody(String.class);
  newEx.getIn().setBody(oldBody+newBody);
  return newEx;
 }
}
```

4. Press **Ctrl+S** to save your bean.

Dropping and linking the components



1. From the **Palette**, expand the **Messaging** folder, and drop a **cFile** component onto the design workspace.

2. Expand the **Routing** folder, and drop a **cAggregate** component onto the design workspace.

3. Expand the **Processor** folder, and drop two **cProcessor** components onto the design workspace.

4. Right-click the **cFile** component, select **Row** > **Route** from the contextual menu and click the first **cProcessor** component.

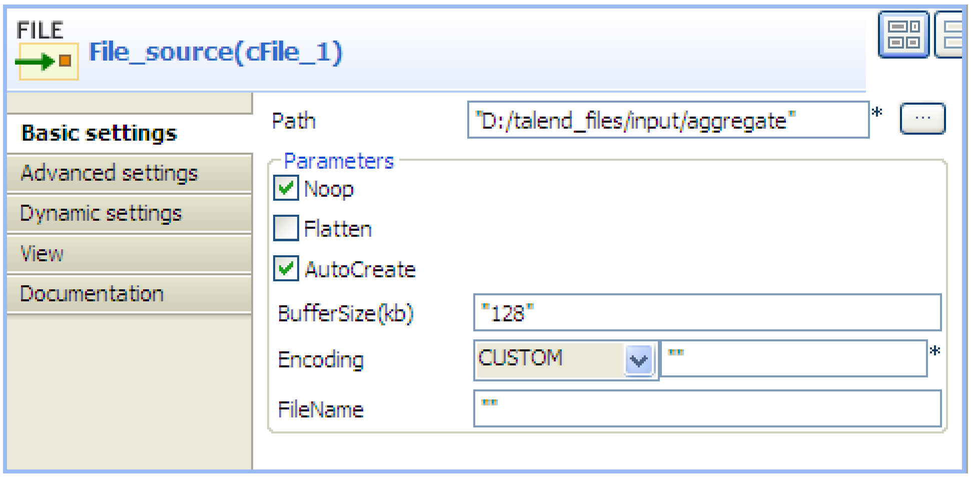
5. Repeat this operation to connect the first **cProcessor** component to the **cAggregate** component.

6. Right-click the **cAggregate** component, select **Row** > **Aggregate** from the contextual menu and click the second **cProcessor** component.

7. Label all the components to better identify their functionality, as shown above.

Configuring the components

1. Double-click the **cFile** component, which is labelled *File_source*, to display its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/input/aggregate" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

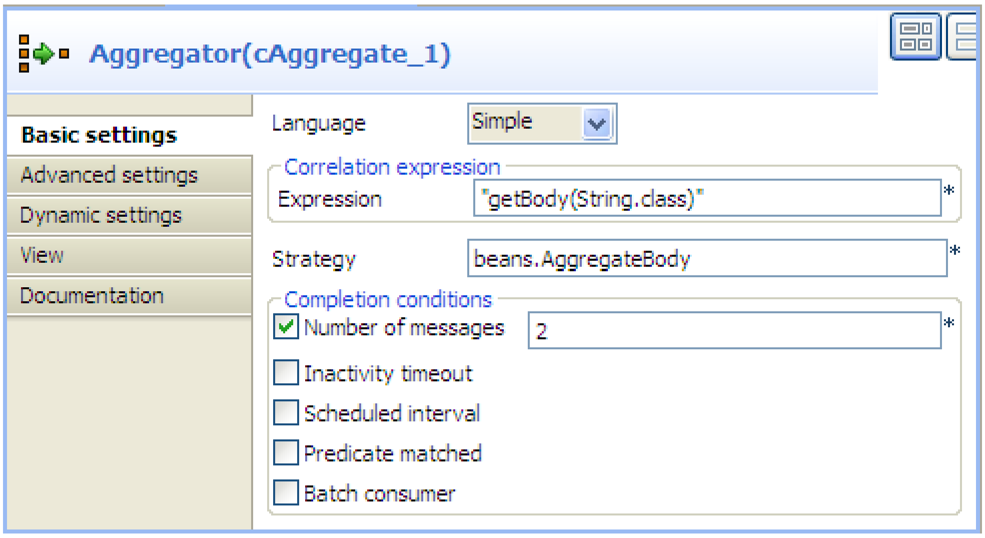
Encoding CUSTOM "" *

FileName ""

2. In the **Path** field, browse to or enter the input file path, and leave the other parameters as they are.

   In this scenario, there are four text files in the specified directory: *a.txt*, *b.txt*, *c.txt* and *d.txt*, the contents of which are *This is a!* , *This is b!* , *This is c!* , and *This is d!* respectively.

3. Double-click the **cAggregate** component, which is labelled *Aggregator*, to display its **Basic settings** view in the **Component** tab.



Language: Simple

Correlation expression

Expression: "getBody(String.class)" *

Strategy: beans.AggregateBody *

Completion conditions

☑ Number of messages: 2 *

□ Inactivity timeout

□ Scheduled interval

□ Predicate matched

□ Batch consumer

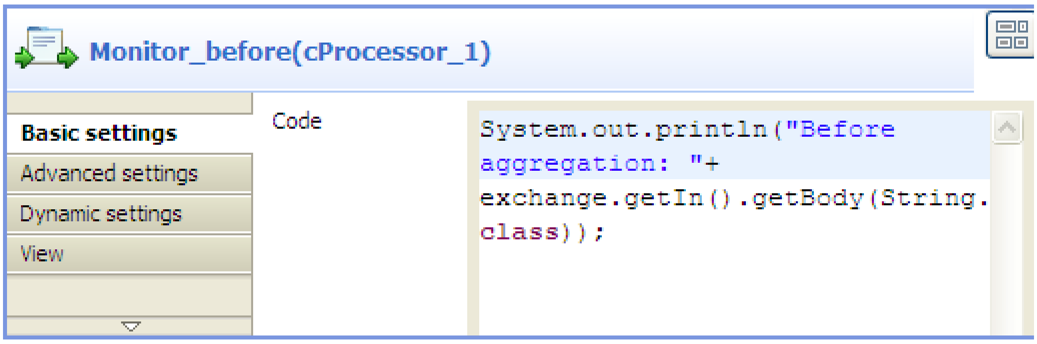
4. In the **Language** field, select **Constant** or **Simple** as the expression language.

   In the **Expression** field, enter the expression `"getBody(String.class)"` to retrieve the body of the message.

   In the **Strategy** field, enter the name of the Java bean *AggregateBody* you just created.

   Select the **Number of messages** check box and type in *2* in the field.

5. Double-click the **cProcessor** component labelled *Monitor_before* to display its **Basic settings** view in the **Component** tab.



Code

```
System.out.println("Before
aggregation: "+
exchange.getIn().getBody(String.
class));
```

6. In the **Code** box, customize the code as follows so that the **Run** console displays the message contents before an aggregation takes place:

```
System.out.println("Before aggregation: "+
exchange.getIn().getBody(String.class));
```

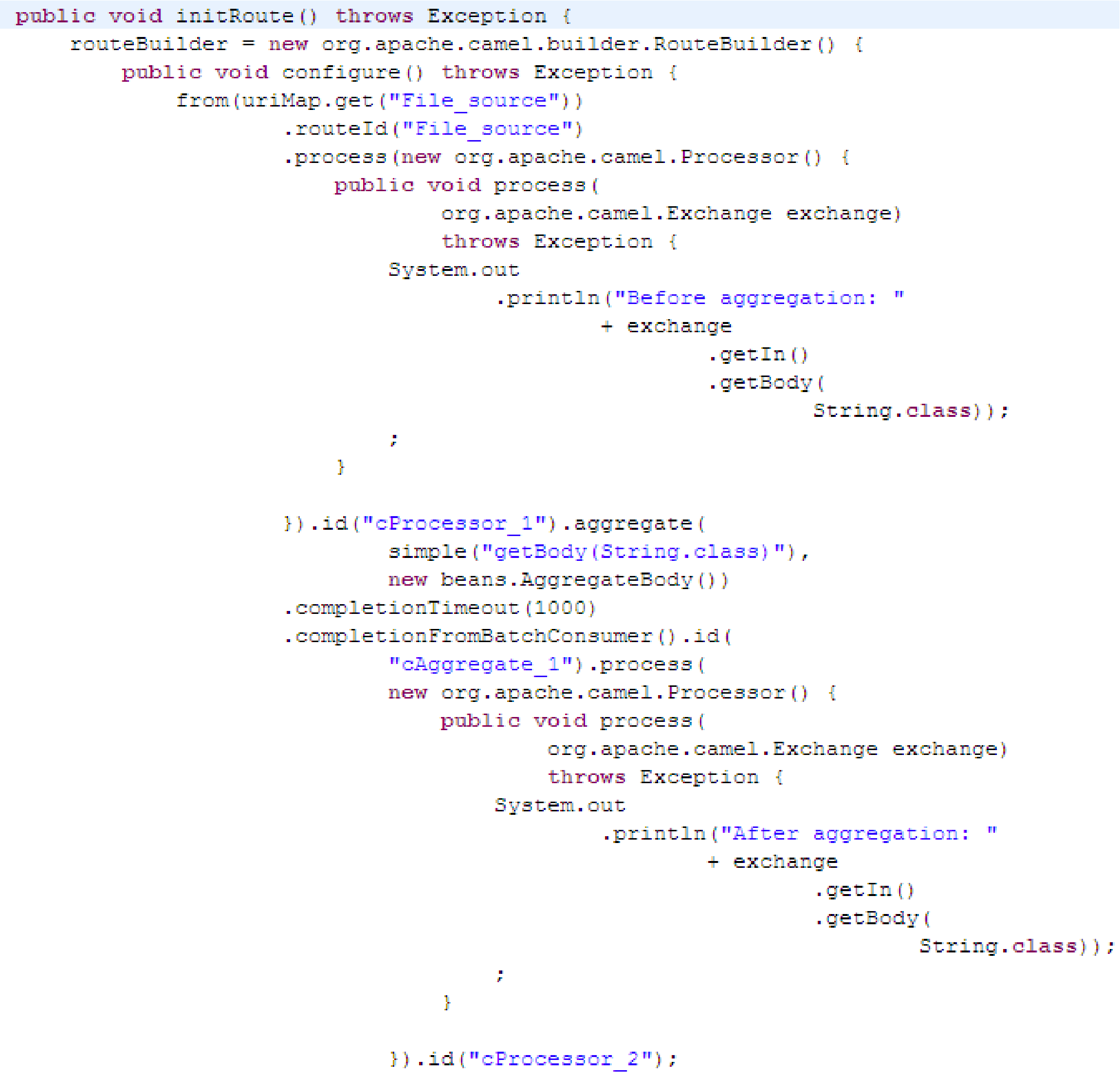
7. In the same way, configure the **cProcessor** component labelled *Monitor_after* so that the Run console displays the message contents after an aggregation takes place:

```
System.out.println("After aggregation: "+
exchange.getIn().getBody(String.class));
```

8. Press **Ctrl+S** to save your route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

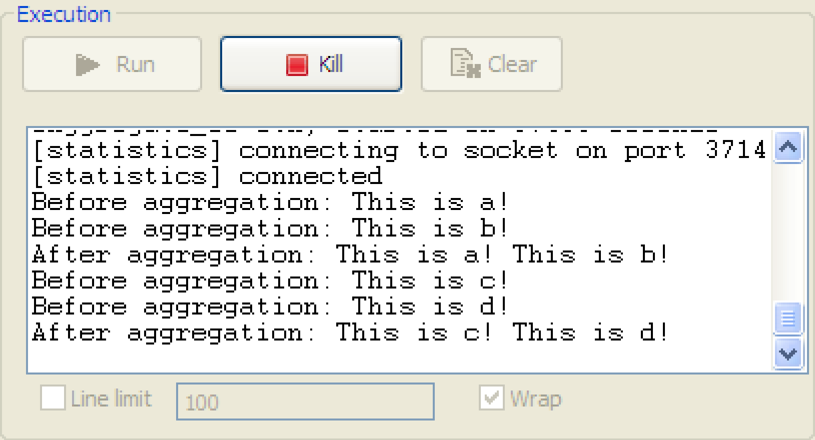


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("File_source"))
                    .routeId("File_source")
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out
                                    .println("Before aggregation: "
                                            + exchange
                                                    .getIn()
                                                    .getBody(
                                                            String.class));
                            ;
                        }

                    }).id("cProcessor_1").aggregate(
                            simple("getBody(String.class)"),
                            new beans.AggregateBody())
                    .completionTimeout(1000)
                    .completionFromBatchConsumer().id(
                            "cAggregate_1").process(
                            new org.apache.camel.Processor() {
                                public void process(
                                        org.apache.camel.Exchange exchange)
                                        throws Exception {
                                    System.out
                                            .println("After aggregation: "
                                                    + exchange
                                                            .getIn()
                                                            .getBody(
                                                                    String.class));
                                    ;
                                }

                            }).id("cProcessor_2");
```

As shown in the code, a message `from` the `File_source` endpoint is routed via `cProcessor_1` and then aggregated according to the condition `.aggregate`.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your route. You can also press **F6** to execute it.

   RESULT: The four messages are aggregated in two batches, two messages combined into one each batch.



```
[statistics] connecting to socket on port 3714
[statistics] connected
Before aggregation: This is a!
Before aggregation: This is b!
After aggregation: This is a! This is b!
Before aggregation: This is c!
Before aggregation: This is d!
After aggregation: This is c! This is d!
```

cDynamicRouter

cDynamicRouter properties

Component Family	Routing	
Function	**cDynamicRouter** allows you to route messages while avoiding the dependency of the router on all possible destinations.	
Purpose	**cDynamicRouter** is used to route a message or messages to different endpoints on specified conditions.	
Basic settings	*Bean class*	Enter the name of the bean class to be used for the dynamic router.
	Specify the method	Select this check box to specify the method to be used which is defined in the bean class.
	Ignore Invalid Endpoints	Select this check box to ignore unresolved endpoint URIs. Clear the check box to throw an exception when endpoint URIs are not valid.
Usage	**cDynamicRouter** is used as a middle or end component in a Route.	
Limitation	n/a	

Scenario: Routing files conditionally to different file paths

In this scenario, three file messages containing people information are routed to different endpoints according to the city names they contain.

The following is an extract of the example XML files used in this use case:

Message_1.xml:

```
<person>
  <firstName>Ellen</firstName>
  <lastName>Ripley</lastName>
  <city>Washington</city>
</person>
```

Message_2.xml:

```
<person>
  <firstName>Peter</firstName>
  <lastName>Green</lastName>
  <city>London</city>
</person>
```

Message_3.xml:

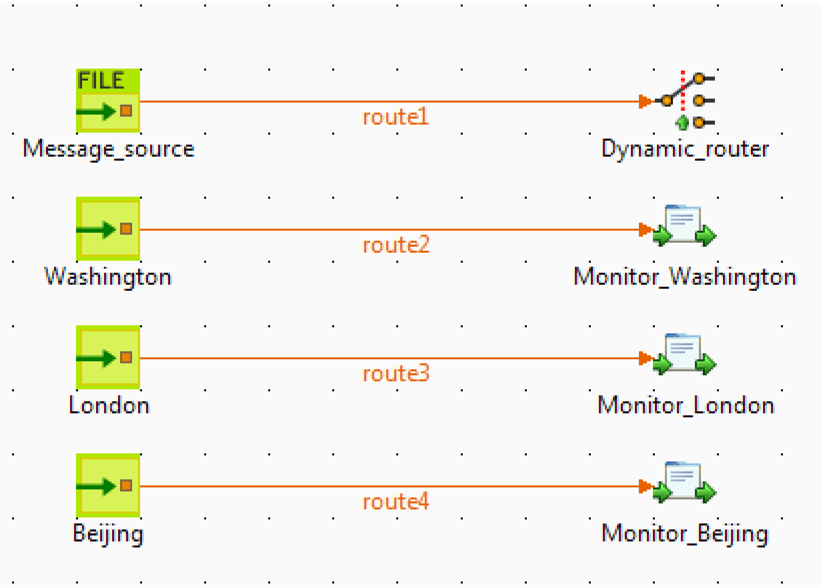
```
<person>
  <firstName>Alice</firstName>
  <lastName>Yang</lastName>
  <city>Beijing</city>
</person>
```

A predefined Java bean, *setDynaURI*, is called in this use case to return endpoint URIs according to the city name contained in each message, so that the message containing the city name *Washington* will be routed to endpoint *Washington* and so forth.

For more information about creating and using Java Beans, see *Talend Open Studio for ESB* **User Guide**.

```
package beans;

import org.apache.camel.Exchange;
import org.apache.camel.Header;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class setDynaURI {

 public String setURI(Document document,
      @Header(Exchange.SLIP_ENDPOINT) String previous) {
    if(previous!=null){
     return null;
    }
  NodeList cities = document.getDocumentElement().getElementsByTagName(
    "city");
  Element city = (Element) cities.item(0);
  String textContent = city.getTextContent();
   return "direct:"+textContent;
  }
}
```

Dropping and linking the components



1. From the **Palette**, expand the **Messaging** folder, and drop one **cFile** and three **cMessagingEndpoint** components onto the design workspace.

2. Expand the **Routing** folder, and drop a **cDynamicRouter** component onto the design workspace.

3. Expand the **Processor** folder, and drop three **cProcessor** components onto the design workspace.

4. Label the components for better identification of their respective functionality.

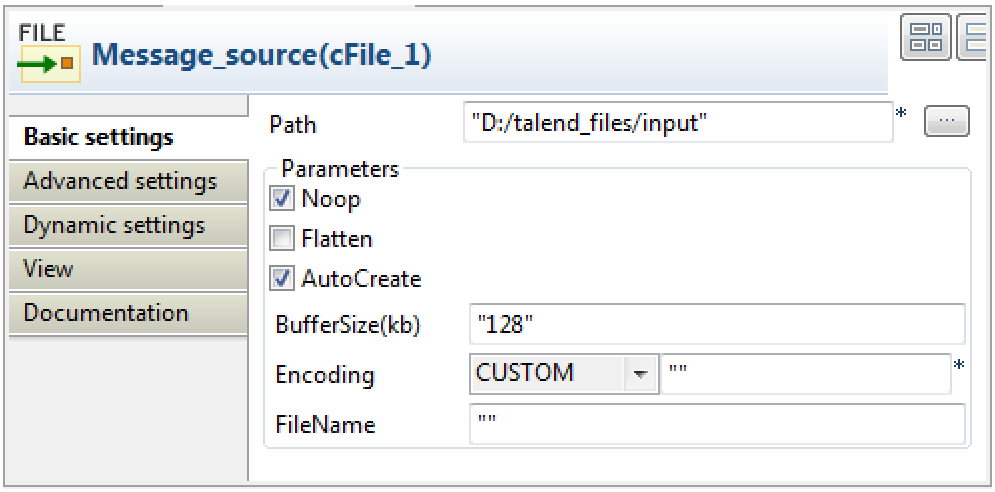
5. Right-click the **cFile** component, select **Row** > **Route** from the contextual menu and click the **cDynamicRouter** component.

6. Repeat this operation to connect the **cMessagingEndpoint** components to the **cProcessor** components.

Configuring the components and connections

1. Double-click the input **cFile** component to display its **Basic settings** view in the **Component** tab and set its properties.

   In this use case, simply specify the input file path and leave the other parameters as they are.



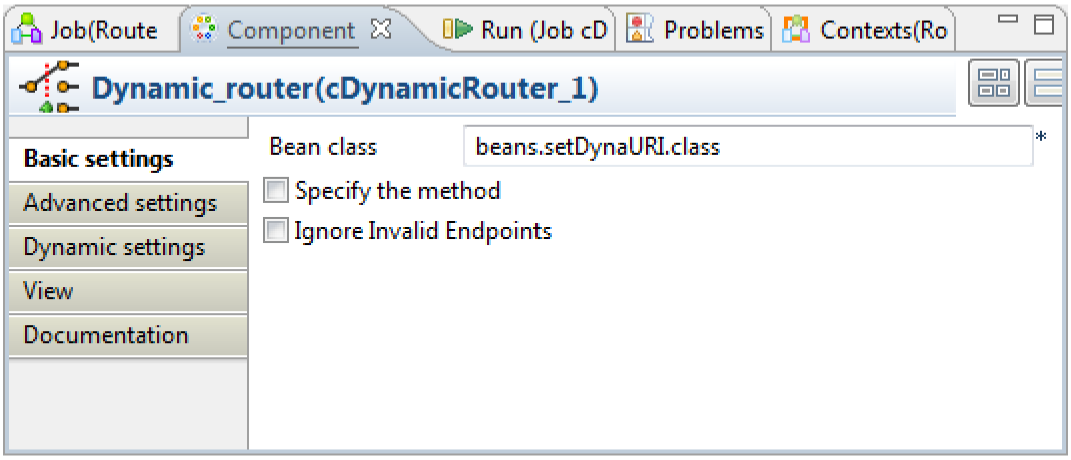
FILE
Message_source(cFile_1)

Path "D:/talend_files/input"

Parameters
☑ Noop
□ Flatten
☑ AutoCreate
BufferSize(kb) "128"
Encoding CUSTOM ""
FileName ""

2. Double-click the **cDynamicRouter** component to display its **Basic settings** view in the **Component** tab.

3. In the **Bean class** field, type in the name of the predefined Java bean. Leave the **Specify the method** check box unselected as there is only one method in the Java bean and leave the **Ignore Invalid Endpoints** check box unselected if you want the component to throw an exception when endpoint URIs are not valid.



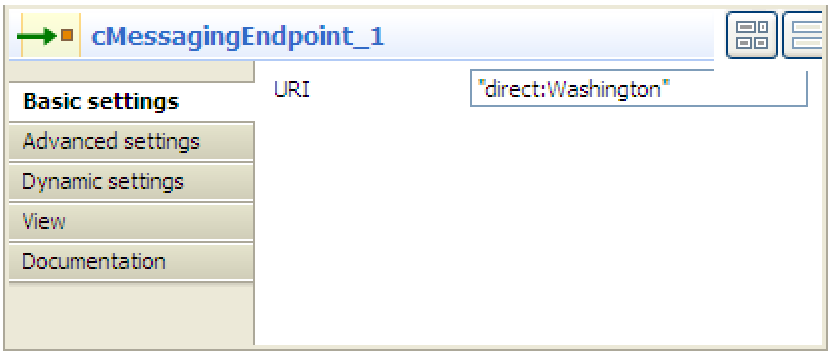
Dynamic_router(cDynamicRouter_1)

Bean class beans.setDynaURI.class

□ Specify the method
□ Ignore Invalid Endpoints

4. Double-click the first **cMessagingEndpoint** component, which is labelled *Washington*, to display its **Basic settings** view in the **Component** tab, and type in the URI in the **URI** field for the destination of your message.

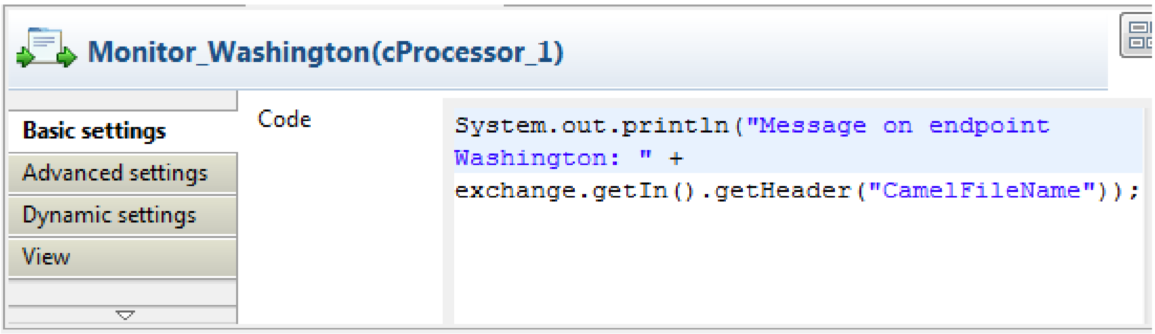
   Here, we want to use this component to retrieve the message routed to the URI *direct:Washington*, as shown below.



URI "direct:Washington"

5. Repeat this step to set the endpoint URIs for the other two **cMessagingEndpoint** components: *direct:London* and *direct:Beijing* respectively.

6. Double-click the first **cProcessor** component, which is labelled *Monitor_Washington*, to display its **Basic settings** view in the **Component** tab.



Code

```
System.out.println("Message on endpoint
Washington: " +
exchange.getIn().getHeader("CamelFileName"));
```

7. In the **Code** box, customize the code to display the file name of the message routed to the endpoint *Washington* on the console.

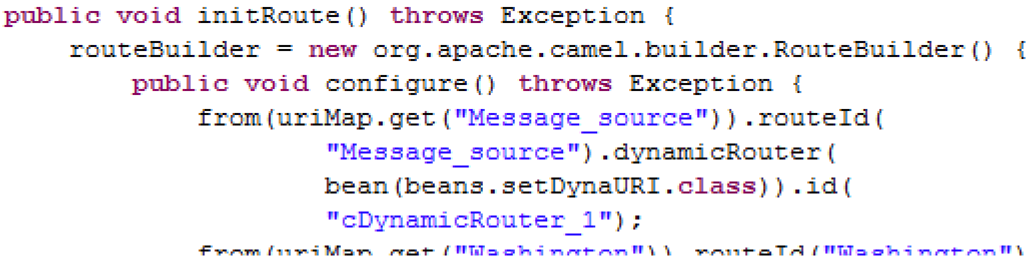
```
System.out.println("Message on endpoint Washington: "+
exchange.getIn().getHeader("CamelFileName"));
```

8. Repeat these steps to configure the other two **cProcessor** components to display the file names of the messages routed to the endpoints *London* and *Beijing* respectively.

9. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.



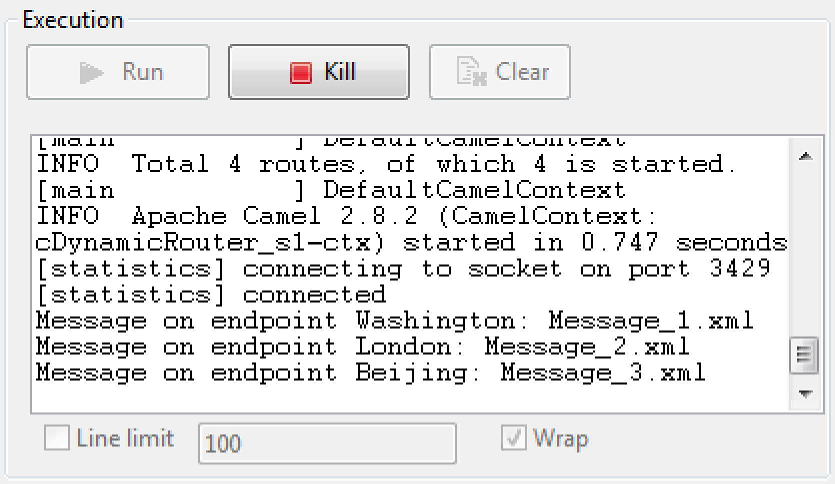
```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Message_source")).routeId(
                    "Message_source").dynamicRouter(
                    bean(beans.setDynaURI.class)).id(
                    "cDynamicRouter_1");
            from(uriMap.get("Washington")).routeId("Washington")
```

As shown in the code, the incoming message `from` the endpoint `Message_source` is routed by `.dynamicRouter` to endpoints the URIs of which are dynamically set according to `beans.setDynaURI.class`.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route.

   You can also press **F6** to execute it.

   RESULT: The source messages are routed to different endpoints based on the city names contained in the messages.



```
[main                ] DefaultCamelContext
INFO  Total 4 routes, of which 4 is started.
[main                ] DefaultCamelContext
INFO  Apache Camel 2.8.2 (CamelContext:
cDynamicRouter_s1-ctx) started in 0.747 seconds
[statistics] connecting to socket on port 3429
[statistics] connected
Message on endpoint Washington: Message_1.xml
Message on endpoint London: Message_2.xml
Message on endpoint Beijing: Message_3.xml
```

cIdempotentConsumer

cIdempotentConsumer properties

Component Family	Routing	
Function	**cIdempotentConsumer** deduplicates messages and thereby prevents the receiving message endpoint from receiving duplicate messages.	
Purpose	**cIdempotentConsumer** identifies messages that have already been sent to the receiver and eliminates them. Messages are still sent by the sender but are ignored by the receiver at the delivery stage.	
Basic settings	*Repository Type*	Message identifiers need to be stored in a repository. For new incoming messages, identifiers are checked against the ones stored in the repository to identify and drop duplicates. There are two ways to store them: **Memory**: messages identifiers are stored temporarily. ⚠ *The in-memory storage mode can easily run out of memory and does not work in a clustered environment.* **File**: messages identifiers are stored in a file. Specify the path to this file in the **File store** field.
	File store	Specify the path and name of the file storing messages identifiers.
	Cache Size	Type in the size of the cache, namely the number of message identifiers to store.
	Use language	Select this check box if you want to specify the language used in the **Predicate** field to specify the identifier of the messages.
	Expression	Type in the expression to use to specify the identifier of the messages.
	Eager	Select this check box to detect duplicate messages even when messages are currently in progress; clear it to detect duplicates only when messages have successfully been processed. By default, this check box is selected.
	SkipDuplicate	Select this check box to drop duplicates; clear it to ignore duplicates so that all messages will be continued. By default, this check box is selected.
Usage	**cIdempotentConsumer** is used as a middle component in a Route.	
Connections	*idemp*	The **idemp** link retrieves messages deduplicated by the **cIdempotentConsumer** component.
	Route	As an optional link, the **Route** link retrieves all messages from the message sender.

Limitation	n/a

Scenario: Deduplicating messages while routing them

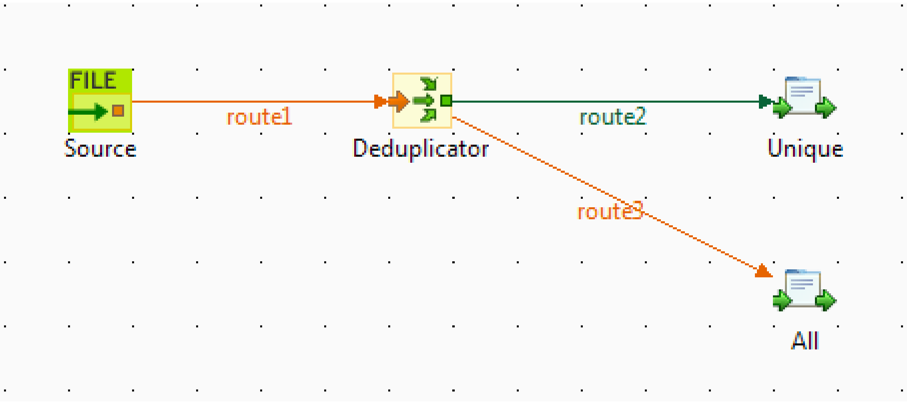
In this scenario, duplicated messages are filtered and only the unique one is routed to the destination.

Three XML files that have the same content, as shown below, are used in this use case.

```
<people>
 <person id="8">
  <firstName>Ellen</firstName>
  <lastName>Ripley</lastName>
  <city>Washington</city>
 </person>
</people>
```

Dropping and linking the components

This use case requires one **cFile** component, one **cIdempotentComsumer** component, and two **cProcessor** components.



1. From the **Palette**, expand the **Messaging** folder, select the **cFile** component, and drop it onto the design workspace as the message source component.

2. Expand the **Routing** folder, select the **cIdempotentComsumer** component and drop it onto the design workspace as the message deduplicator.

3. Expand the **Processor** folder, drop two **cProcessor** components onto the design workspace, one as the consumer for deduplicated messages and another for all messages.

4. Right-click the **cFile** component, select **Row** > **Route** from the contextual menu and click the **cIdempotentComsumer** component.

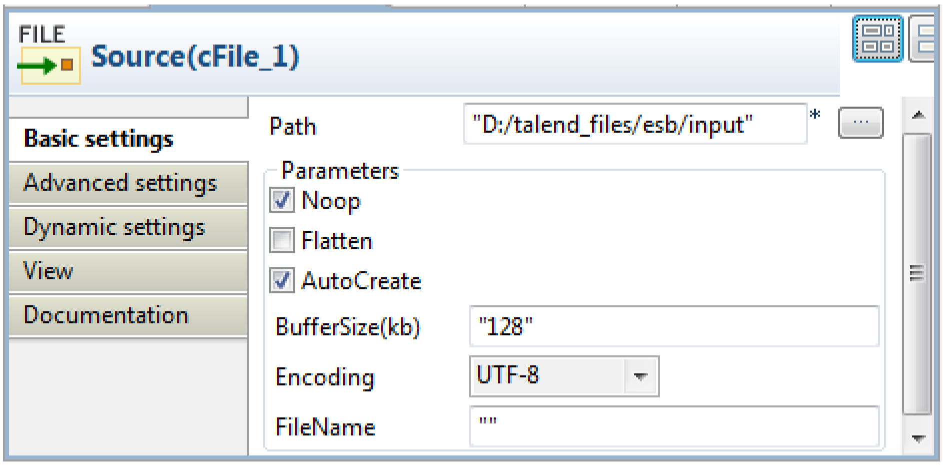
5. Right-click the **cIdempotentComsumer** component, select **Row** > **idemp** from the contextual menu and click the **cProcessor** component on the top.

6. Connect the **cIdempotentComsumer** component to the other **cProcessor** component using a **Row** > **Route** connection. This optional connection will retrieve all the messages coming from the source.

7. Label the components to better identify their roles in the Route.

Configuring the components and connections

1. Double-click the **cFile** component, which is labelled *Source*, to display its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/esb/input" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

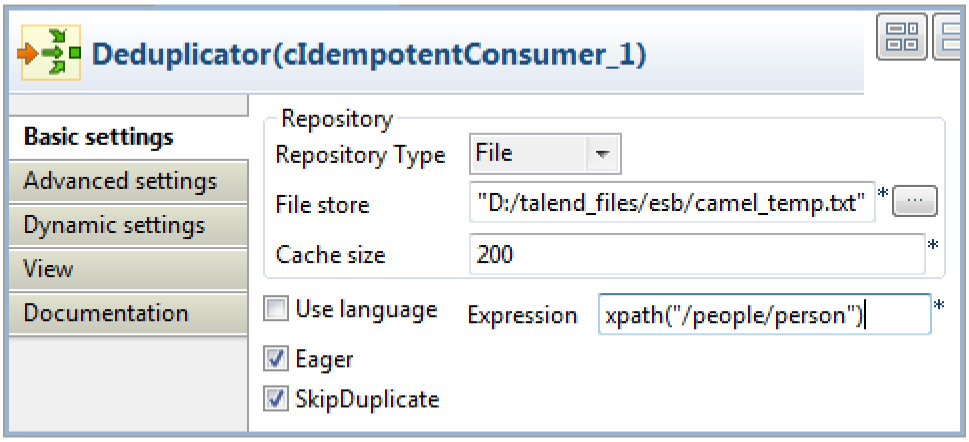
Encoding UTF-8

FileName ""

2. In the **Path** field, specify the file path to the message source.

   From the **Encoding** list, select the encoding type of your source files, and leave all the other parameters as they are.

3. Double-click the **cIdempotentComsumer** component, which is labelled *Deduplicator*, to display its **Basic settings** view in the **Component** tab.



Repository

Repository Type File

File store "D:/talend_files/esb/camel_temp.txt" *

Cache size 200 *

□ Use language Expression xpath("/people/person") *

☑ Eager

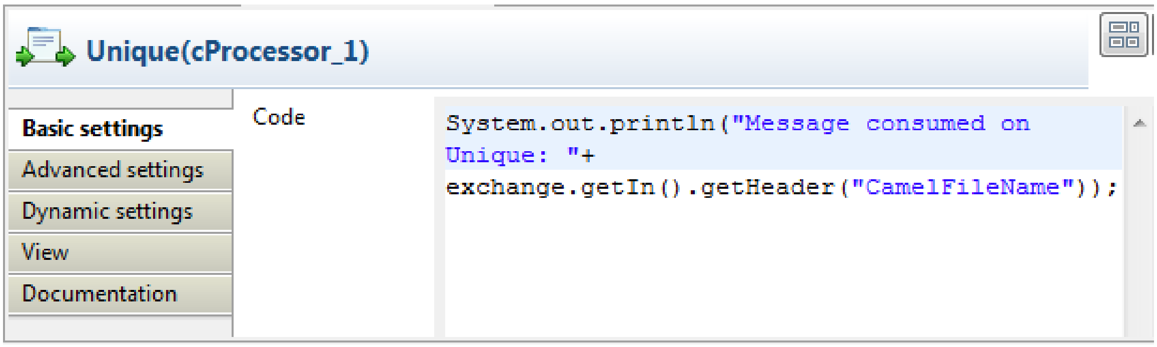
☑ SkipDuplicate

4. From the **Repository Type** list, select between **Memory** and **File** to specify where the message identifiers will be stored before the deduplication process. For this scenario, select **File**.

   In the **File store** field, specify the location of the file storing message identifiers.

   In the **Expression** field, enter an expression to filter the messages. In this scenario, enter the following expression to filter the messages according to the *person* node of the XML files: `xpath("/people/person")`, and leave all the other parameters as they are. Alternatively, you can select the **Use language** check box, select **XPath** from the **Language** list, and enter `"/people/person"` in the **Predicate** field.

5. Double-click the **cProcessor** component labelled *Unique* to display its **Basic settings** view in the **Component** tab.



Code

```
System.out.println("Message consumed on
Unique: "+
exchange.getIn().getHeader("CamelFileName"));
```

6. In the **Code** area, customize the code to display the file name of the message that passes the deduplication:

```
System.out.println("Message consumed on Unique: "+
exchange.getIn().getHeader("CamelFileName"));
```

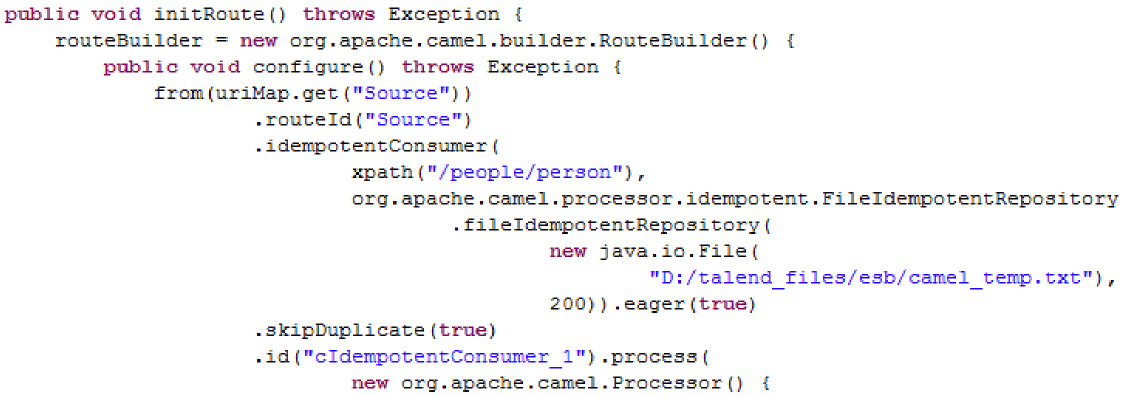
7. Repeat these steps to configure the other **cProcessor** component, which is labelled *All*, to display the file names of all the messages coming from the source:

```
System.out.println("Message consumed on All: "+
exchange.getIn().getHeader("CamelFileName"));
```

8. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to view the generated code.

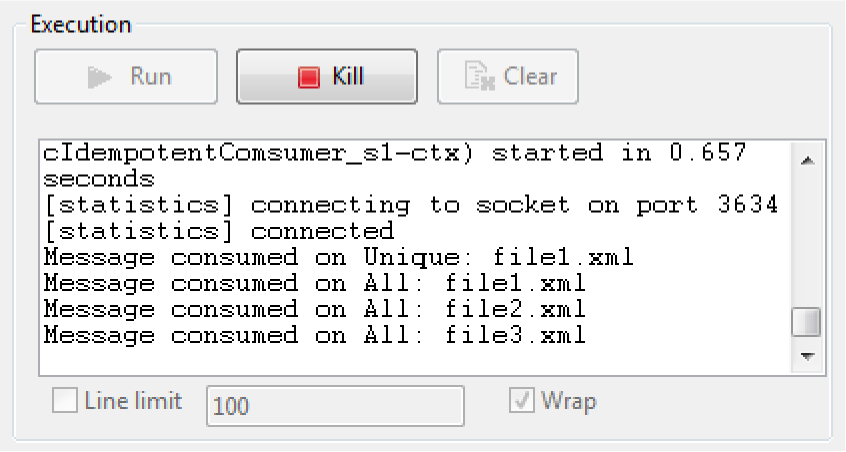


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Source"))
                    .routeId("Source")
                    .idempotentConsumer(
                            xpath("/people/person"),
                            org.apache.camel.processor.idempotent.FileIdempotentRepository
                                    .fileIdempotentRepository(
                                            new java.io.File(
                                                    "D:/talend_files/esb/camel_temp.txt"),
                                            200)).eager(true)
                    .skipDuplicate(true)
                    .id("cIdempotentConsumer_1").process(
                            new org.apache.camel.Processor() {
```

In this partially shown piece of code, messages `from` the `Source` are filtered according to the expression `xpath("/people/person")` and deduplicated by `cIdempotentConsumer_1`.

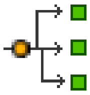
2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route. You can also press **F6** to execute it.

RESULT: When several files have the same content, only the first one is routed to the receiving endpoint.



```
cIdempotentComsumer_s1-ctx) started in 0.657
seconds
[statistics] connecting to socket on port 3634
[statistics] connected
Message consumed on Unique: file1.xml
Message consumed on All: file1.xml
Message consumed on All: file2.xml
Message consumed on All: file3.xml
```

cLoadBalancer



cLoadBalancer properties

Component Family	Routing	
Function	**cLoadBalancer** allows you to distribute messages across multiple endpoints using different load balancing strategies.	
Purpose	**cLoadBalancer** allows you to distribute messages among several endpoints using a variety of load balancing strategies.	
Basic settings	*Strategy*	Select between **Random**, **Round Robin**, **Sticky**, **Topic**, **Failover**, and **Custom**. Each method is described below.
Random	The receiving endpoint is chosen randomly at each exchange.	
Round Robin	Messages are distributed according to the round robin method which distributes the load evenly.	
Sticky	*Language*	Select the language of the expression to use in the **Expression** field to distribute the messages.
	Expression	Type in the expression that will be used to calculate a correlation key that will determine the endpoint to choose.
Topic	Select this option to send all the messages to all the endpoints.	
Failover	*Basic mode*	By default, the failover load balancing always sends the messages to the first endpoint. If the first endpoint fails, the messages are sent to subsequent endpoints.
	Specify exceptions	Specify the exceptions to which the failover should react to in the **Exception** table.
	Use with Round robin	Select this option to use failover with advanced options. From the **Maximum failover attempt** list, select the number of attempt to be proceed before giving up the transfer: -**Attempt forever**: always attempts to transfer the messages and always try to failover. -**Never failover**: gives up immediately the transfer of messages and never try to failover. -**A number of attempts**: attempts *n* number of time to transfer messages, specify that number in the **Number of attempts** field. **Inherit error handler**: Select *true* if you want Camel error handler to be used. If you select *false*, the load balancer will immediately failover when an exception is thrown. **Use Round robin**: Select *true* if you want to combine failover with round robin. Failover load balancing with round robin mode distributes the load evenly between the services, and it provides automatic failover.
Custom	*Load balancer*	Type in the name of your custom load balancer.

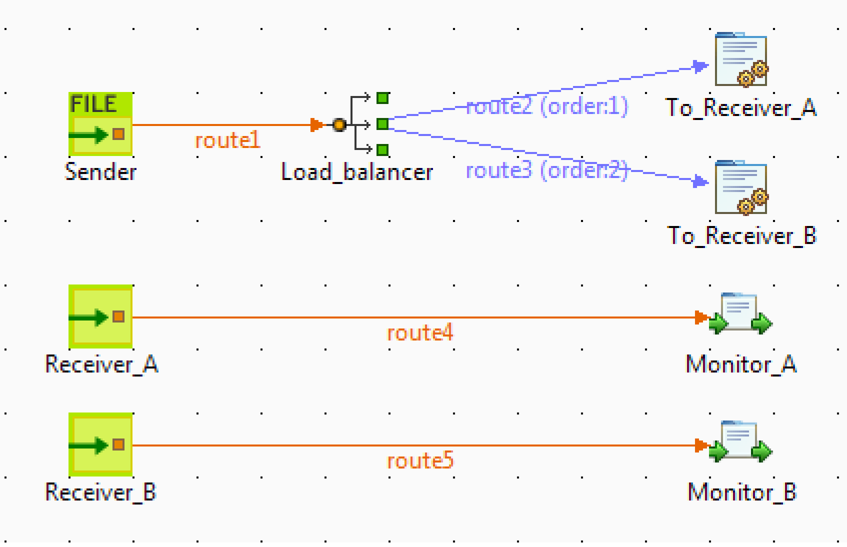
Usage	**cLoadBalancer** is used as a middle component in a Route.	
Connections	*Load Balance*	Select this link to route messages to the next endpoint according to the selected load-balancing strategy.
	Route	Select this link to route all the messages from the sender to the next endpoint.
Limitation	n/a	

Scenario: Distributing messages to receiver endpoints based on round robin

In this scenario, a **cLoadBalancer** component is used to distribute four messages evenly to two receiving endpoints in accordance with the round robin load balancing method.

Dropping and linking the components

This scenario requires one **cFile** component as the message sender, one **cLoadBalancer** component to distribute the messages to two different receivers in a load balancing manner, two **cJavaDSLProcessor** components to define the URIs of the receivers, two **cMessagingEndpoint** components to retrieve the messages routed to the two receivers, and two **cProcessor** components to display the effect of round robin load balancing.



1. From the **Messaging** folder of the **Palette**, drop one **cFile** component and two **cMessagingEndpoint** components onto the workspace, and label them according to their roles in the Route: *Sender*, *Receiver_A*, and *Receiver_B* respectively.

2. From the **Routing** folder, drop a **cLoadBalancer** component onto the design workspace, and label it *Load_balancer*.

3. From the **Processor** folder, drop two **cJavaDSLProcessor** components and two **cProcessor** components onto the design workspace, and label them according to their roles in the Route: *To_Receiver_A*, *To_Receiver_B*, *Monitor_A*, and *Monitor_B* respectively.

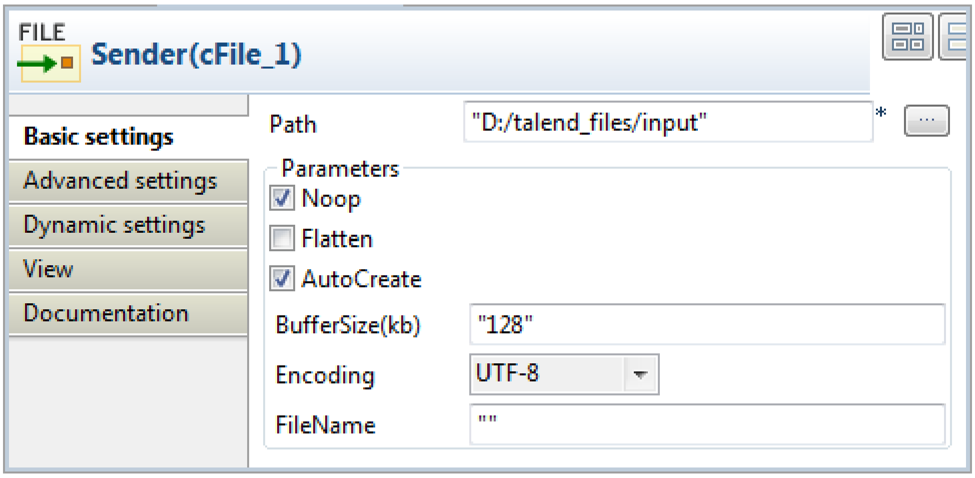
4. Link the **cFile** component to the **cLoadBalancer** component using a **Row** > **Route** connection.

5. Link **cLoadBalancer** to each of the two **cJavaDSLProcessor** components using a **Row** > **Load Balance** connection.

6. Link each of the two **cMessagingEndpoint** components to the corresponding **cProcessor** component using a **Row** > **Route** connection.

Configuring the components and connections

1. Double-click the **cFile** component to open its **Basic Settings** view in the **Component** tab.



Path "D:/talend_files/input" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

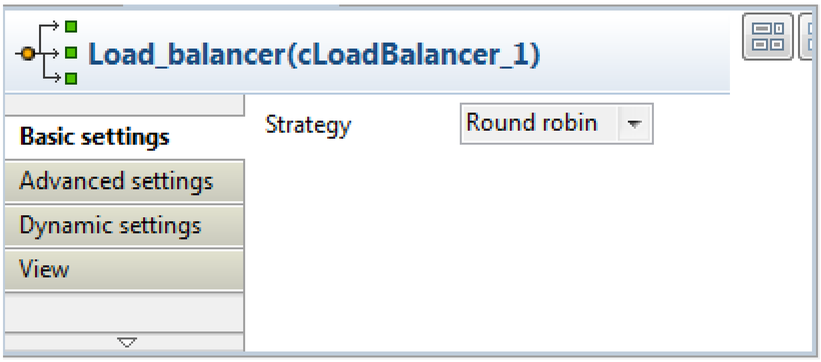
Encoding UTF-8

FileName ""

2. In the **Path** field, specify the file path to message source.

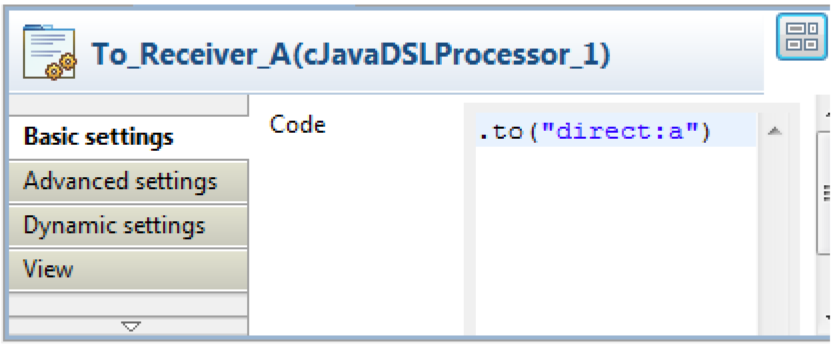
3. From the **Encoding** list, select the encoding type of your message files. Leave the other parameters as they are.

4. Double-click the **cLoadBalancer** component to open its **Basic Settings** view in the **Component** tab, and select the load balancing method you want to use from the **Strategy** list. In this scenario, we use the default **Round robin** method.



Strategy Round robin

5. Double-click the **cJavaDSLProcessor** component labeled *To_Receiver_A* to open its **Basic Settings** view in the **Component** tab, and enter URI of the first receiver between the double quotation marks in the **Code** area, `direct:a` in this example.

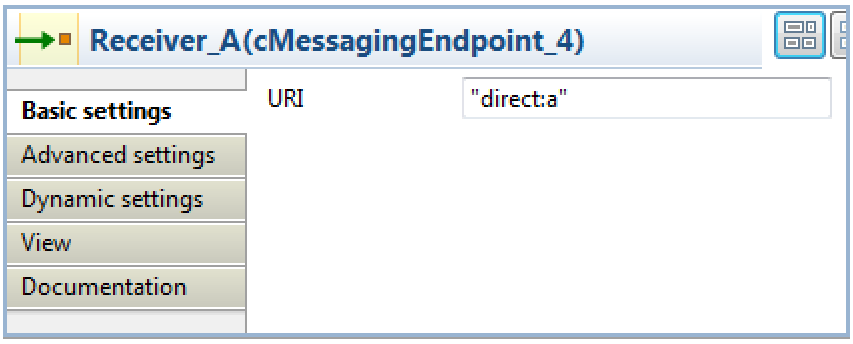


Code

```
.to("direct:a")
```

Repeat this step to define the URI of the other receiver, `direct:b`, in the **cJavaDSLProcessor** component labeled *To_Receiver_B*.

6. Double-click the **cMessagingEndpoint** component labeled *Receiver_A* to open its **Basic Settings** view in the **Component** tab, and enter URI of the first receiver between the double quotation marks in the **URI** field, `direct:a` in this example.



URI "direct:a"

Repeat this step to define the URI of the other receiver, `direct:b`, in the **cMessagingEndpoint** component labeled *Receiver_B*.

7. Double-click the **cProcessor** component labeled *Monitor_A* to open its **Basic Settings** view in the **Component** tab, and customize the code in the **Code** area to display the file names of the messages routed to *Receiver_A* on the console:

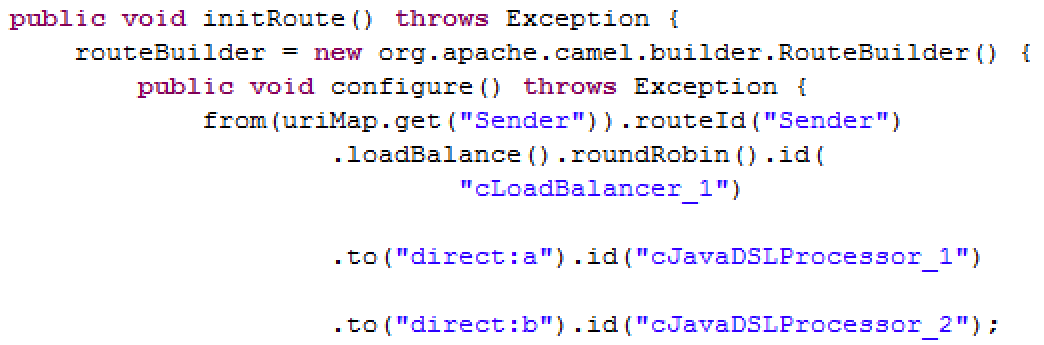
```
System.out.println("Message on Receiver_A: "+
exchange.getIn().getHeader("CamelFileName"));
```

Repeat this step to customize the code in the cProcessor component labeled Monitor_B to display the file names of the messages routed to *Receiver_B* on the console.

8. Press **Ctrl+S** to save your Route.

Viewing the code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to check the generated code:

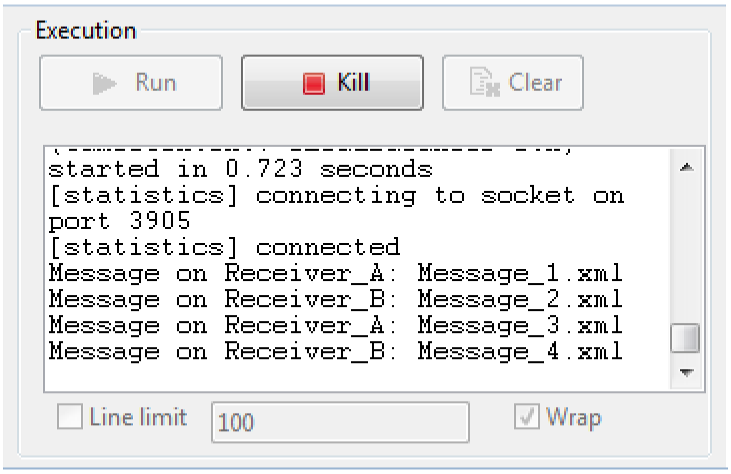


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender")).routeId("Sender")
                    .loadBalance().roundRobin().id(
                            "cLoadBalancer_1")

                    .to("direct:a").id("cJavaDSLProcessor_1")

                    .to("direct:b").id("cJavaDSLProcessor_2");
```

As shown above, while messages are routed `from` the source endpoint `.to` the destination endpoints, routing load balancing is implemented according to the `.roundRobin()` method by `cLoadBalancer_1`.

2. Press **F6** to run your Route.

RESULT: Of the four messages from the sender, two are routed to *Receiver_A* and two are routed to *Receiver_B* in a round robin manner.



```
started in 0.723 seconds
[statistics] connecting to socket on
port 3905
[statistics] connected
Message on Receiver_A: Message_1.xml
Message on Receiver_B: Message_2.xml
Message on Receiver_A: Message_3.xml
Message on Receiver_B: Message_4.xml
```

cMessageFilter

cMessageFilter properties

Component Family	Routing	
Function	**cMessageFilter** filters the content of messages according to the specified criterion and routes the filtered messages to the specified output channel. All messages that do not match the criteria will be dropped. For more information on the Camel Message Filter EIP: http://camel.apache.org/message-filter.html.	
Purpose	Use **cMessageFilter** to eliminate unwanted messages from a channel according to the defined criterion.	
Basic settings	*Language*	Select the language of the expression you use to filter your messages from **Constant**, **EL**, **Groovy**, **Header**, **JavaScript**, **JoSQL**, **JXPath**, **MVEL**, **None**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**.
	Expression	Type in the expression to use to filter the messages.
Usage	**cMessageFilter** is used as a middle component in a Route.	
Connections	*Filter*	Select this link to route the filtered messages to the next endpoint.
	Route	Select this link to route all the messages from the sender to the next endpoint.
Limitation	n/a	

Scenario: Filtering messages according to a criterion

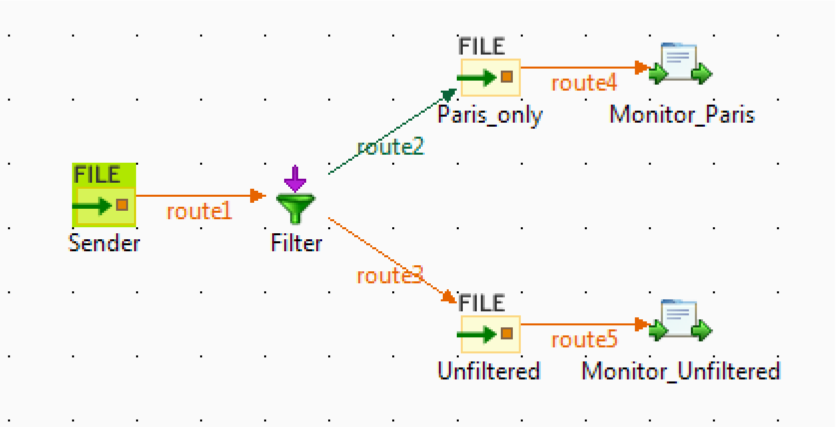
In this use case, we filter XML messages that are sent from the sending endpoint according to a defined criterion: only the XML files in which the value of the *city* node is *Paris* are sent to a folder named *Paris_only*.

Of the four XML files used in this scenario, *Message_1.xml* and *Message_4.xml* contain the city name of *Paris*. The following is an example:

```
<person>
  <firstName>Pierre</firstName>
  <lastName>Dupont</lastName>
  <city>Paris</city>
</person>
```

Dropping and linking the components

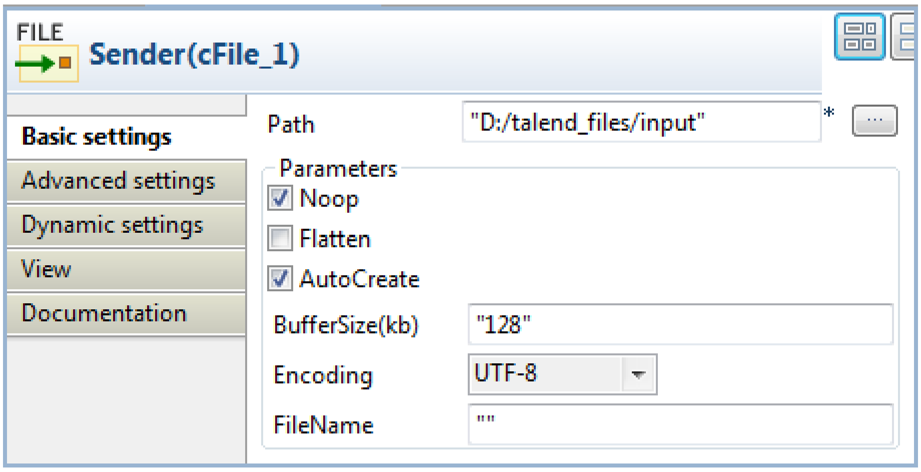
This scenario requires one **cMessageFilter** component to filter the messages from the sender, one **cFile** component as the message sender, one **cFile** component to receiver the messages containing *Paris*, one **cFile** component to receiver all the messages from the sender, and two **cProcessor** components to monitor the messages routed to the two receivers.



1. From the **Messaging** folder of the **Palette**, drop three **cFile** components onto the design workspace, and label them *Sender*, *Paris_only*, and *Unfiltered* respectively to better identify their roles.
2. From the **Routing** folder, drop a **cMessageFilter** component onto the design workspace, and label it *Filter*.
3. From the **Processor** folder, drop two **cProcessor** components onto the design workspace, and label them *Monitor_Paris* and *Monitor_Unfiltered* respectively.
4. Right-click the **cFile** component labeled *Sender*, select **Row** > **Route** from the contextual menu and click the **cMessageFilter** component.
5. Right-click the **cMessageFilter** component, select **Row** > **Filter** from the contextual menu and click the **cFile** component labeled *Paris_only*. This endpoint will retrieve the messages that meet the defined criterion.
6. Right-click the **cMessageFilter** component, select **Row** > **Route** from the contextual menu and click the **cFile** component labeled *Unfiltered*. This endpoint will collect all the messages, including those meeting the filter criterion. This connection is optional.
7. Right-click the **cFile** component labeled *Paris_only*, select **Row** > **Route** from the contextual menu and click the **cProcessor** component labeled *Monitor_Paris*. Repeat this step to connect the **cFile** component labeled *Unfiltered* to the **cProcessor** component labeled *Monitor_Unfiltered*.

Configuring the components and connections

1. Double-click the **cFile** component labeled *Sender* to open its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/input" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

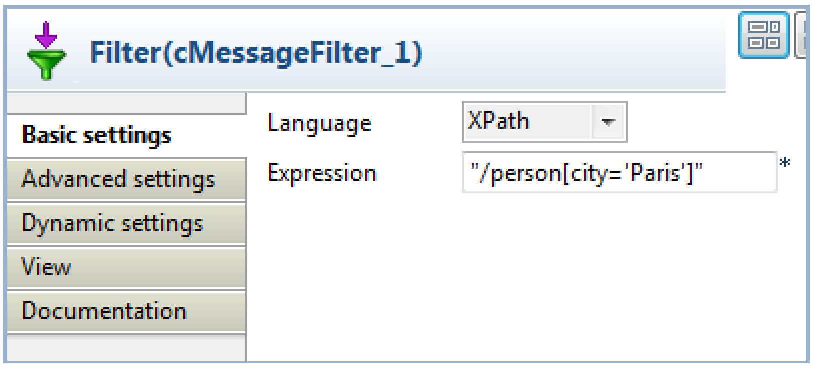
BufferSize(kb) "128"

Encoding UTF-8

FileName ""

2. In the **Path** field, specify the file path to message source.
3. From the **Encoding** list, select the encoding type of your message files. Leave the other parameters as they are.

4. Double-click the **cMessageFilter** component to open its **Basic settings** view in the **Component** tab.



Filter(cMessageFilter_1)

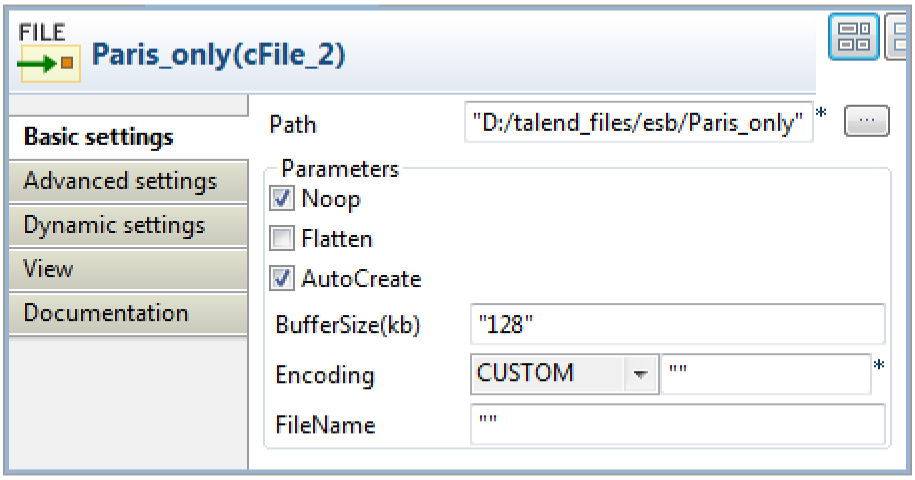
Language: XPath

Expression: "/person[city='Paris']" *

5. Select the language of the expression you want to use to filter your messages, and enter an expression to define a criterion according to which you want to filter your messages.

   In this scenario, we want to sort out the XML files containing a city node with the value of Paris, so we select **XPath** from the **Language** list, and fill the in the **Expression** field with this expression: `"/person[city='Paris']"`.

6. Double-click the **cFile** component labeled *Paris_only* to open its **Basic settings** view in the **Component** view, and specify the path for the messages meeting the filter criterion in the **Path** field.



Paris_only(cFile_2)

Path: "D:/talend_files/esb/Paris_only" *

Parameters

☑ Noop

☐ Flatten

☑ AutoCreate

BufferSize(kb): "128"

Encoding: CUSTOM "" *

FileName: ""

   Repeat this step to define the path for all the messages from the sender in the **cFile** component labeled *Unfiltered*.

7. Double-click the **cProcessor** component labeled *Monitor_Paris* to open its **Basic settings** view in the **Component** view, and customize the code in the **Code** area to display the file names of the messages that meet the filter criterion on the console:

```
System.out.println("Message sent to folder Paris_only: "+
exchange.getIn().getHeader("CamelFileName"));
```

   Repeat this step to customize the code in the **cProcessor** component labeled *Monitor_Unfiltered* to display the file names of all the messages from the sender.

8. Press **Ctrl+S** to save your Route.

Viewing the code and executing the Route

1. To have a look at the generated code, click the **Code** tab at the bottom of the design workspace.

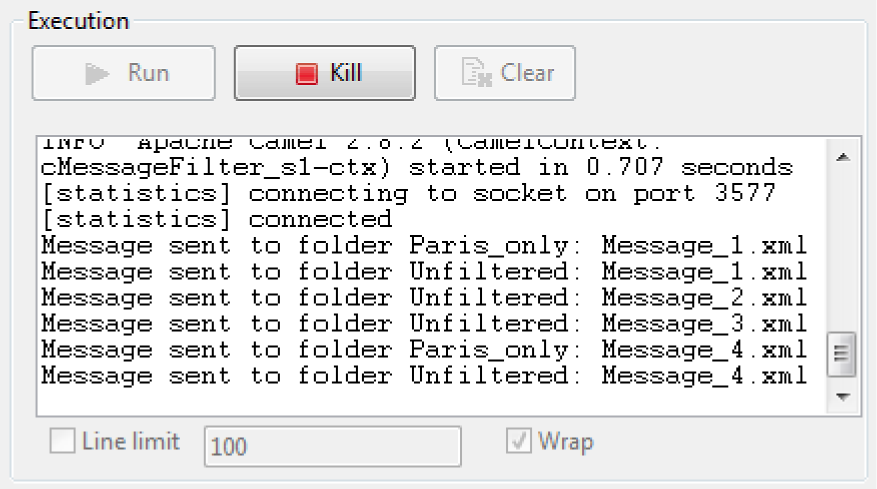
```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender"))
                    .routeId("Sender")
                    .filter()
                    .xpath("/person[city='Paris']")
                    .id("cMessageFilter_1")
                    .to(uriMap.get("Paris_only"))
                    .id("cFile_2")
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out
                                    .println("Message sent to folder Paris_only: "
                                            + exchange
                                                    .getIn()
                                                    .getHeader(
                                                            "CamelFileName"));
                            ;
                        }

                    }).id("cProcessor_1").end().to(
                            uriMap.get("Unfiltered")).id("cFile_3")
```

As shown in this piece of code, messages `from` the sender are filtered by `cMessageFilter_1` according to `.xpath("/person[city='Paris']")` and the messages matching the filter are send `.to` the endpoint `Paris_only`, while all messages are sent `.to` the endpoint `Unfiltered`.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route.

   You can also press **F6** to execute it.

   RESULT: The messages are filtered according to the defined criterion and the messages containing "*Paris*" are redirected to the *Paris_only* folder, all the messages, including those containing "*Paris*", are sent to the *Unfiltered* folder.



```
INFO  Apache Camel 2.8.2 (CamelContext:
cMessageFilter_s1-ctx) started in 0.707 seconds
[statistics] connecting to socket on port 3577
[statistics] connected
Message sent to folder Paris_only: Message_1.xml
Message sent to folder Unfiltered: Message_1.xml
Message sent to folder Unfiltered: Message_2.xml
Message sent to folder Unfiltered: Message_3.xml
Message sent to folder Paris_only: Message_4.xml
Message sent to folder Unfiltered: Message_4.xml
```

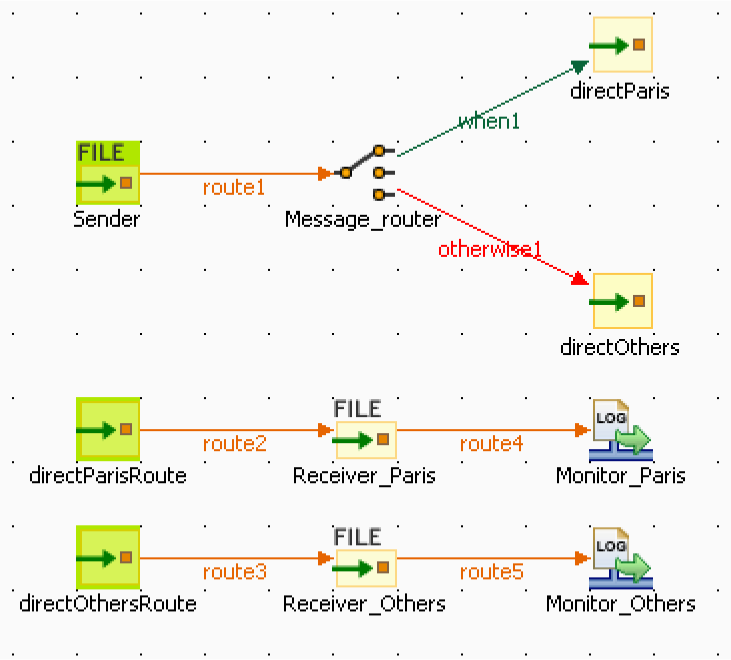
cMessageRouter

cMessageRouter properties

Component Family	Routing	
Function	**cMessageRouter** routes messages in different channels according to specified conditions.	
Purpose	**cMessageRouter** creates different channels for each filtered message types so that messages can later on be treated more accurately in each new channel.	
Usage	**cMessageRouter** is used as a middle component in a Route. It can only have one input channel but multiple output channels. Messages can be outputted through either a **When**, **Otherwise** or **Route** types of connection.	
Connections	*Trigger / When*	Select the **When** link and click the **Component** view. In the **Type** list, select the type of language you will use to declare your condition. In the **Condition** field, type in the condition that will be used to filter the messages. All the messages that do not match this condition are retrieved with the **Otherwise** link to a different channel or dropped if an **Otherwise** link does not present. There can be more than one **When** link in a Route.
	Trigger / Otherwise	This link automatically retrieves the messages that do not match the **When** conditions. There can be only one **Otherwise** link, which is optional, in a Route.
Limitation	It is recommended not to put any message handling after the **When** or the **Otherwise** link. Always use a Mock/Direct endpoint to replace them and make a new Route to handle the messages.	

Scenario: Routing messages according to a criterion

In this use case, we route XML messages that are sent from the sending endpoint according to a defined criterion: those XML files in which the value of the *city* node is *Paris* are sent to a folder named *Paris_only*, and other messages are sent to a folder named *Other_cities*.



Of the four XML files used in this scenario, *Message_1.xml* and *Message_4.xml* contain the city name of *Paris*. The following is an example:

```
<person>
  <firstName>Pierre</firstName>
  <lastName>Dupont</lastName>
  <city>Paris</city>
</person>
```

Dropping and linking the components

1. From the **Messaging** folder of the **Palette**, drop three **cFile** and four **cMessagingEndpoint** components onto the design workspace, and label them *Sender*, *Receiver_Paris*, and *Receiver_Others*, *directParis*, *directOthers*, *directParisRoute*, and *directOthersRoute* respectively to better identify their roles.

2. From the **Routing** folder, drop a **cMessageRouter** component onto the design workspace, and label it *Message_router*.

3. From the **Miscellaneous** folder, drop two **cLog** components onto the design workspace, and label them *Monitor_Paris* and *Monitor_Others* respectively.

4. Right-click the **cFile** component labeled *Sender*, select **Row** > **Route** from the contextual menu and click the **cMessageRouter** component.

5. Right-click the **cMessageRouter** component, select **Trigger** > **When** from the contextual menu and click the **cMessagingEndpoint** component labeled *directParis*. This endpoint will retrieve the messages that meet the defined criterion.

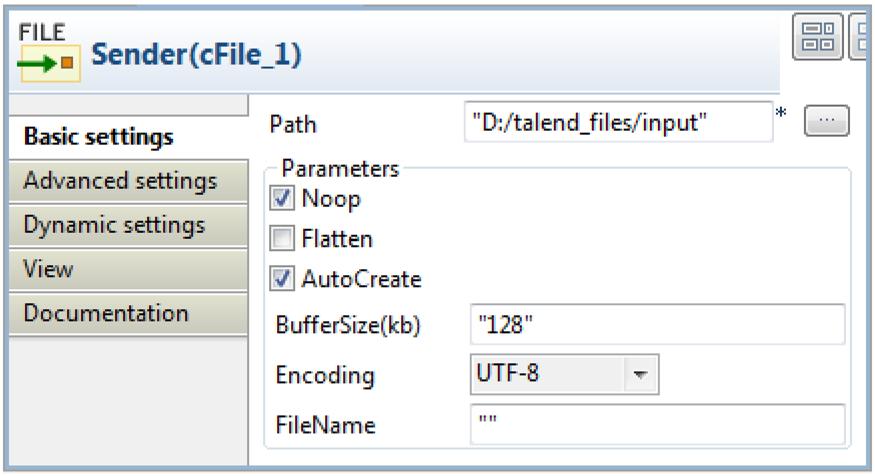
6. Right-click the **cMessageRouter** component, select **Trigger** > **Otherwise** from the contextual menu and click the **cMessagingEndpoint** component labeled *directOthers*. This endpoint will collect all the messages that do not meet the filter criterion.

7. Right-click the **cMessagingEndpoint** component labeled *directParis*, select **Row** > **Route** from the contextual menu and click the **cFile** component labeled *Receiver_Paris*. Repeat this operation to link the component labeled *Receiver_Paris* to *Monitor_Paris*, *directOthersRoute* to *Receiver_Others*, and *Receiver_Others* to *Monitor_Others* respectively using the **Row** > **Route** connection.

Configuring the components and connections

The **cMessageRouter** component does not have any property as it filters and routes the messages from one endpoint to others based on the conditions set in its **When** connection(s).

1. Double-click the **cFile** component labeled *Sender* to open its **Basic settings** view in the **Component** tab.



Sender(cFile_1)

Path "D:/talend_files/input" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

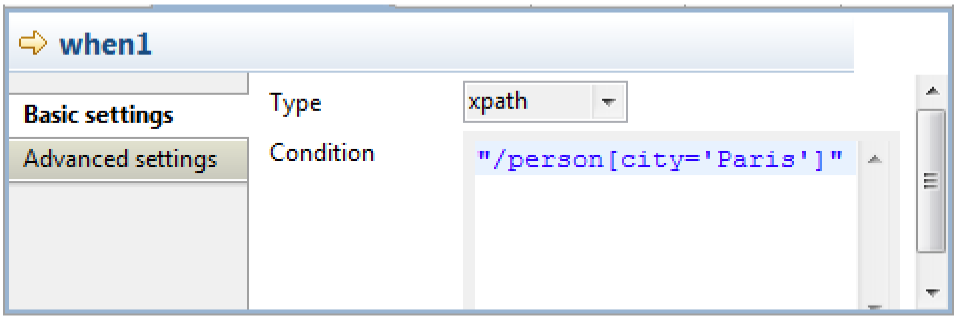
Encoding UTF-8

FileName ""

2. In the **Path** field, specify the file path to message source.

   From the **Encoding** list, select the encoding type of your message files. Leave the other parameters as they are.

3. In the design workspace, click the **When** connection you created and click the **Component** view to define a filter against which messages will be routed.



when1

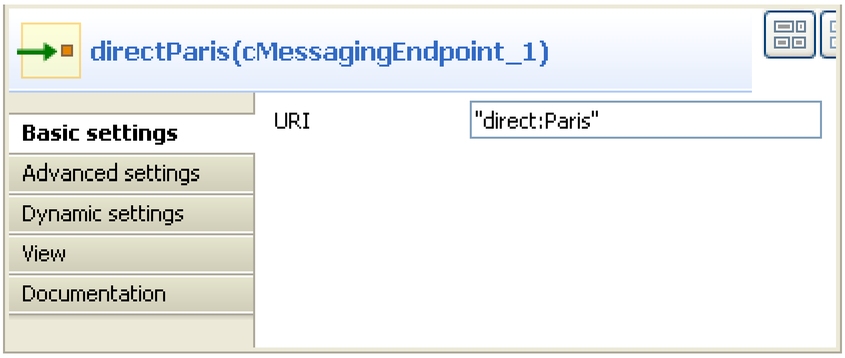
Type xpath

Condition "/person[city='Paris']"

4. In the **Type** list, select **xpath** because the format of the messages used is XML.

   In the **Condition** field, type in `"/person[city='Paris']"` to retrieve only those messages in which the value of the *city* node is *Paris*.

5. Double-click the **cMessagingEndpoint** component labeled *directParis* to open its **Basic settings** view in the **Component** tab.



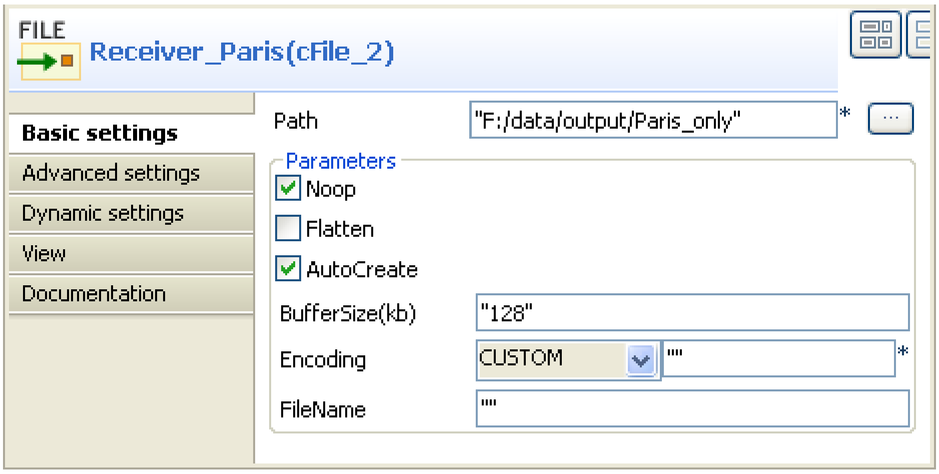
directParis(cMessagingEndpoint_1)

URI "direct:Paris"

6. In the **URI** field, enter the endpoint URI, for example, *"direct:Paris"* to receive the filtered message.

7. Repeat these steps to set the endpoint URI of the **cMessagingEndpoint** components labeled *directOthers* as *"direct:Others"*. Set the endpoint URIs of the **cMessagingEndpoint** components labeled *directParisRoute* and *directOthersRoute* as *"direct:Paris"* and *"direct:Others"* respectively.

8. Double-click the **cFile** component labeled *Receiver_Paris* to open its **Basic settings** view in the **Component** tab, and specify the path for the messages meeting the filter criterion in the **Path** field.



FILE Receiver_Paris(cFile_2)

Path "F:/data/output/Paris_only"

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

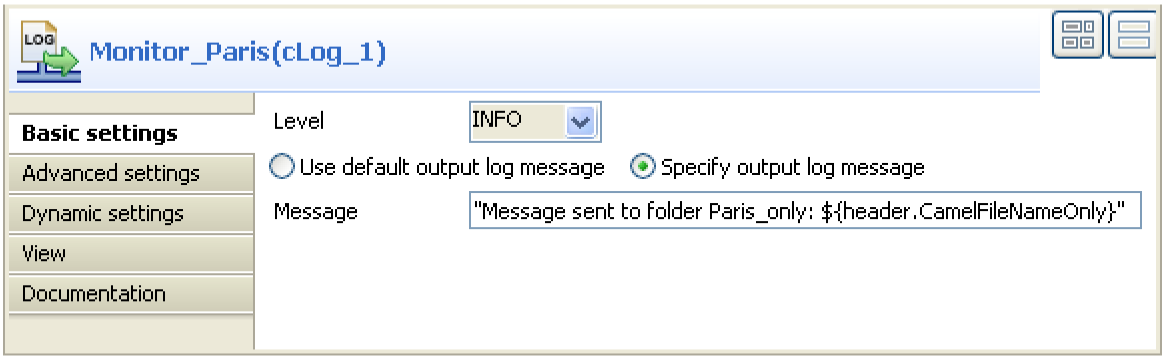
BufferSize(kb) "128"

Encoding CUSTOM ""

FileName ""

Repeat this step to define the path for all the other messages from the sender in the **cFile** component labeled *Receiver_Others*.

9. Double-click the **cLog** component labeled *Monitor_Paris* to open its **Basic settings** view in the **Component** tab.



Monitor_Paris(cLog_1)

Level INFO

○ Use default output log message ● Specify output log message

Message "Message sent to folder Paris_only: ${header.CamelFileNameOnly}"

10. Select **INFO** in the **Level** list. Select the **Specify output log message** option and enter the following code in the **Message** field to display the filename of the message sent to the specified directory.

```
Message sent to folder Paris_only: ${header.CamelFileNameOnly}
```

Repeat this step to customize the message in the **cLog** component labeled *Monitor_Others* to display the filename of the message sent to the specified directory.

11. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

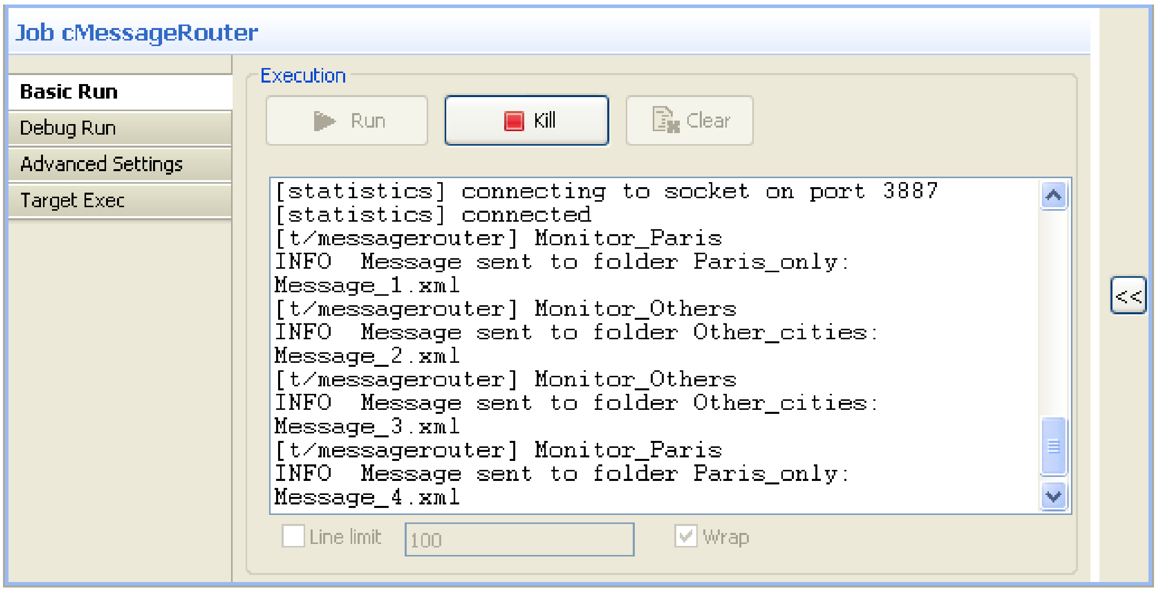
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender")).routeId("Sender").choice()
                    .id("cMessageRouter_1").when().xpath(
                            "/person[city='Paris']").to(
                            uriMap.get("directParis")).id(
                            "cMessagingEndpoint_1").otherwise().to(
                            uriMap.get("directOthers")).id(
                            "cMessagingEndpoint_2");
            from(uriMap.get("directParisRoute"))
                    .routeId("directParisRoute")
                    .to(uriMap.get("Receiver_Paris"))
                    .id("cFile_2")
                    .log(org.apache.camel.LoggingLevel.INFO,
                            "Monitor_Paris",
                            "Message sent to folder Paris_only: ${header.CamelFileNameOnly}")

                    .id("cLog_1");
            from(uriMap.get("directOthersRoute"))
                    .routeId("directOthersRoute")
                    .to(uriMap.get("Receiver_Others"))
                    .id("cFile_3")
                    .log(org.apache.camel.LoggingLevel.INFO,
                            "Monitor_Others",
                            "Message sent to folder Other_cities: ${header.CamelFileNameOnly}")

                    .id("cLog_2");
```

As shown in the code, the messages are routed according to conditions initialized with the `.choice()` piece of code. The filter you defined is initialized with the `.when()` piece of code, and the non filtered messages are routed through the `.otherwise()` piece of code.

2. Click the **Run** button in the **Run** view or press **F6** to execute your Route.

   RESULT: The files containing “*Paris*” are sent to a folder named *Paris_only*, and the other messages are sent in a folder called *Other_cities*.



```
[statistics] connecting to socket on port 3887
[statistics] connected
[t/messagerouter] Monitor_Paris
INFO  Message sent to folder Paris_only:
Message_1.xml
[t/messagerouter] Monitor_Others
INFO  Message sent to folder Other_cities:
Message_2.xml
[t/messagerouter] Monitor_Others
INFO  Message sent to folder Other_cities:
Message_3.xml
[t/messagerouter] Monitor_Paris
INFO  Message sent to folder Paris_only:
Message_4.xml
```

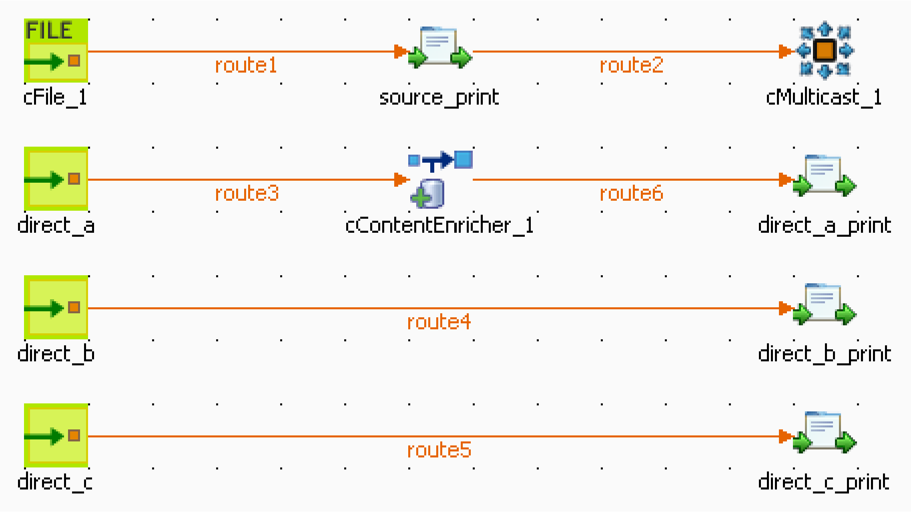
cMulticast

cMulticast properties

Component Family	Routing	
Function	**cMulticast** routes one or more messages to a number of endpoints at one go.	
Purpose	**cMulticast** is used to route one or more messages to a number of endpoints at one go and process them in different ways.	
Basic settings	*URIS*	Add as many lines as needed in the URIs table to define the endpoints to route the message(s) to.
	Send in parallel	Select this check box to multicast the message(s) to the specified endpoints simultaneously.
	Set timeout	Select this check box and set a timeout in the **Timeout** field, in milliseconds. If **cMulticast** fails to send and process all the messages within the set timeframe, it breaks out and continues. Note that this check box appears only when the **Send in parallel** check box is selected.
	Use Aggregation Strategy	Select this check box to refer to a predefined Java bean as an aggregation strategy for assembling the messages from the message source into a single outgoing message. By default, the last message acts as the outgoing message.
Usage	**cMulticast** can be used as a middle or end component in a Route.	
Limitation	n/a	

Scenario: Multicasting a message to two endpoints and using it to enrich the contents received by the third endpoint

In this scenario, a **cMulticast** component is used to route a message to two endpoints. Afterwards, that message is added to the contents received by the third endpoint by using a **cContentEnricher** component.



To build the Route, do the following.

Dropping and linking the components

1. Drag and drop the following components from the **Palette** onto the workspace: one **cFile**, three **cMessagingEndpoint**, four **cProcessor**, one **cMulticast** and one **cContentEnricher**. For better identification of the components' functionalities, change the labels of the three **cMessagingEndpoint** components to **direct_a**, **direct_b** and **direct_c**, and change the labels of the four **cProcessor** components to **source_print**, **direct_a_print**, **direct_b_print** and **direct_c_print**.

2. Link **cFile** to **source_print** using a **Row** > **Route** connection.

3. Link **source_print** to **cMulticast** using a **Row** > **Route** connection.

4. Link **direct_a** to **cContentEnricher** using a **Row** > **Route** connection.

5. Link **cContentEnricher** to **direct_a_print** using a **Row** > **Route** connection.

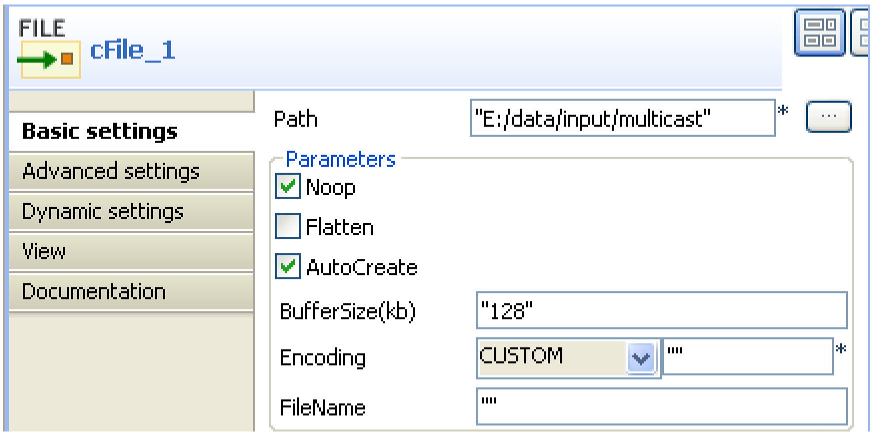
6. Link **direct_b** to **direct_b_print** using a **Row** > **Route** connection.

7. Link **direct_c** to **direct_c_print** using a **Row** > **Route** connection.

Configuring the components

Procedure 1. Configuring the data source and the multicast recipients

1. Double-click **cFile** to open its **Basic settings** view in the **Component** tab.



FILE cFile_1

Path "E:/data/input/multicast"

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

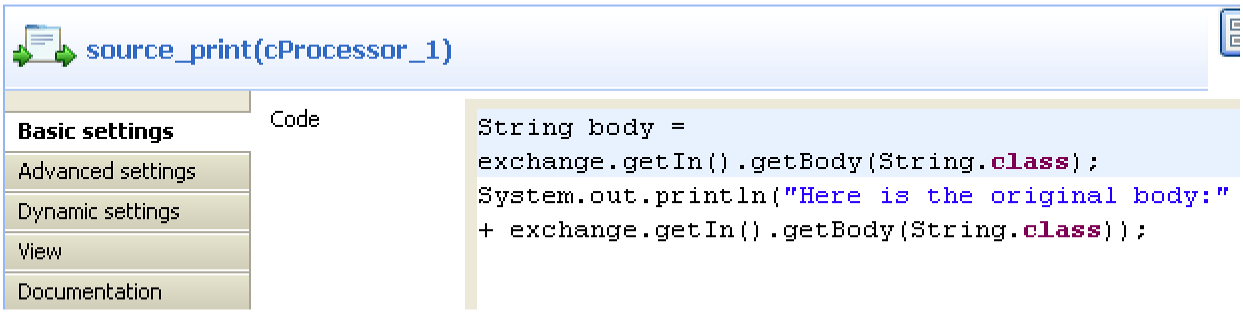
BufferSize(kb) "128"

Encoding CUSTOM ""

FileName ""

2. In the **Path** field, type in the path to the source message, for example, *"E:/data/input/multicast"*. Keep the default settings for other fields.

3. Double-click **source_print** to open its **Basic settings** view in the **Component** tab.



source_print(cProcessor_1)

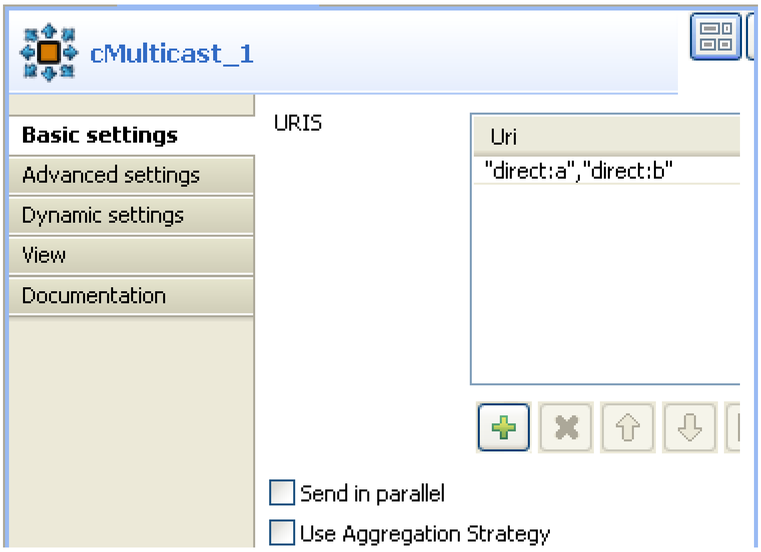
Code

```
String body =
exchange.getIn().getBody(String.class);
System.out.println("Here is the original body:"
+ exchange.getIn().getBody(String.class));
```

4. In the **Code** box, enter the code below to get the source message body and print it out.

```
String body = exchange.getIn().getBody(String.class);
System.out.println("Here is the original body:"
+ exchange.getIn().getBody(String.class));
```

5. Double-click **cMulticast** to open its **Basic settings** view in the **Component** tab.



cMulticast_1

URIS

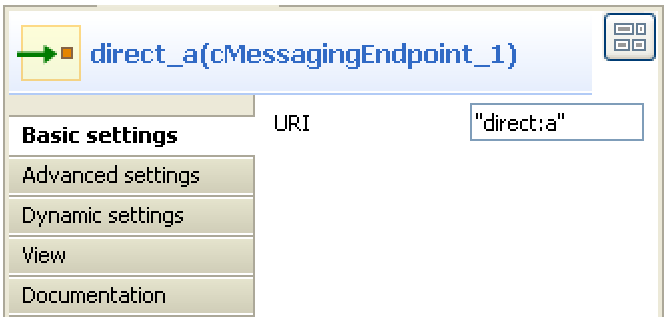
Uri
"direct:a","direct:b"

□ Send in parallel

□ Use Aggregation Strategy

6. In the **URIS** table, click the plus button to add a line where you need to type in the URIs of the endpoints to receive the multicast message, for example, *"direct:a","direct:b"*.

7. Double-click **direct_a** to open its **Basic settings** view in the **Component** tab.



direct_a(cMessagingEndpoint_1)

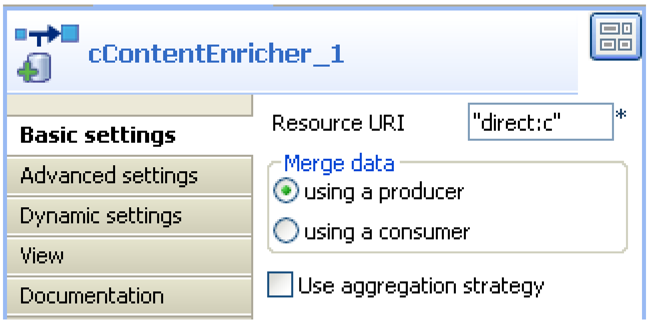
URI "direct:a"

8. In the **URI** field, enter the endpoint URI, for example, *"direct:a"*.

   Perform the same operation to **direct_b** and **direct_c** and type in the URIs of *"direct:b"* and *"direct:c"* respectively.

Procedure 2. Configuring the content enricher and printers

1. Double-click **cContentEnricher** to open its **Basic settings** view in the **Component** tab.



cContentEnricher_1

Resource URI "direct:c" *

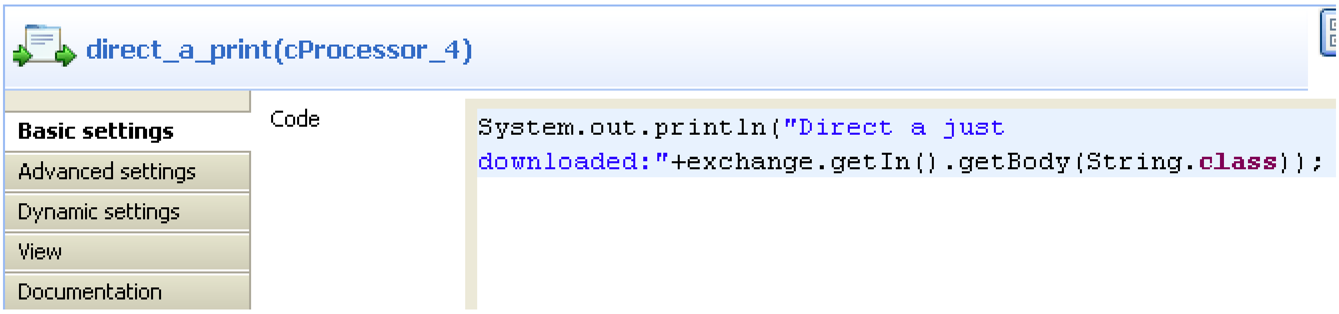
Merge data
● using a producer
○ using a consumer

□ Use aggregation strategy

2. In the **Resource URI** field, type in the URI of the endpoint whose incoming contents will be enriched with the message received by **cContentEnricher**, for example, *"direct:c"*.

   In the **Merge data** area, select **using a producer** to enable **cContentEnricher** to route the received message to *"direct:c"*.

3. Double-click **direct_a_print** to open its **Basic settings** view in the **Component** tab.



direct_a_print(cProcessor_4)

Code

```
System.out.println("Direct a just
downloaded:"+exchange.getIn().getBody(String.class));
```

4. In the **Code** box, enter the code below to print out the message received by **direct_a**.

```
System.out.println("Direct a just
 downloaded:"+exchange.getIn().getBody(String.class));
```

   Perform the same operation to **direct_b_print** and **direct_c_print** and type in the code below in their code boxes in turn:

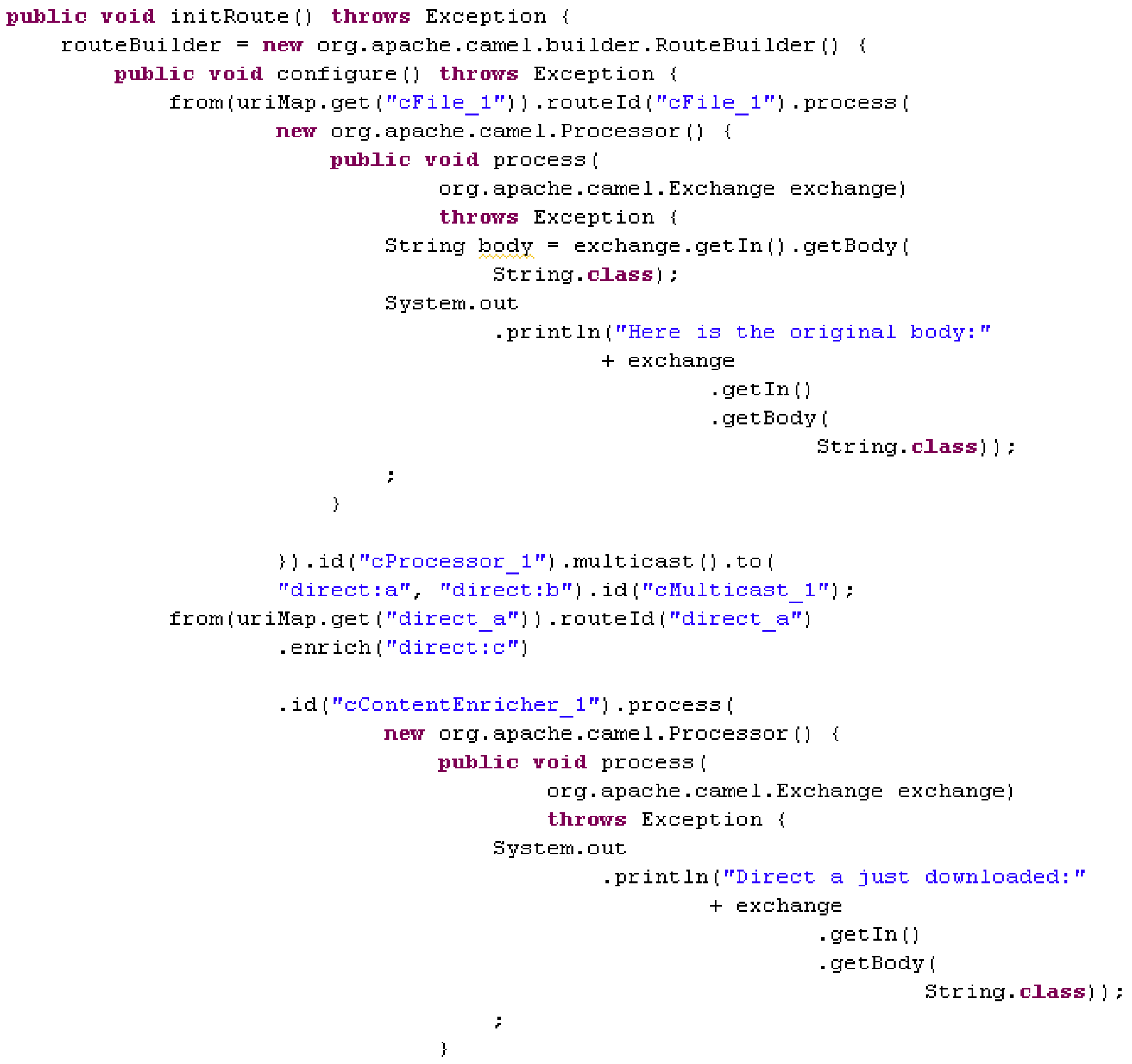
```
System.out.println("Direct b just
 downloaded:"+exchange.getIn().getBody(String.class));
```

```
System.out.println("Direct c just
 downloaded:"+exchange.getIn().getBody(String.class));
```

5. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.

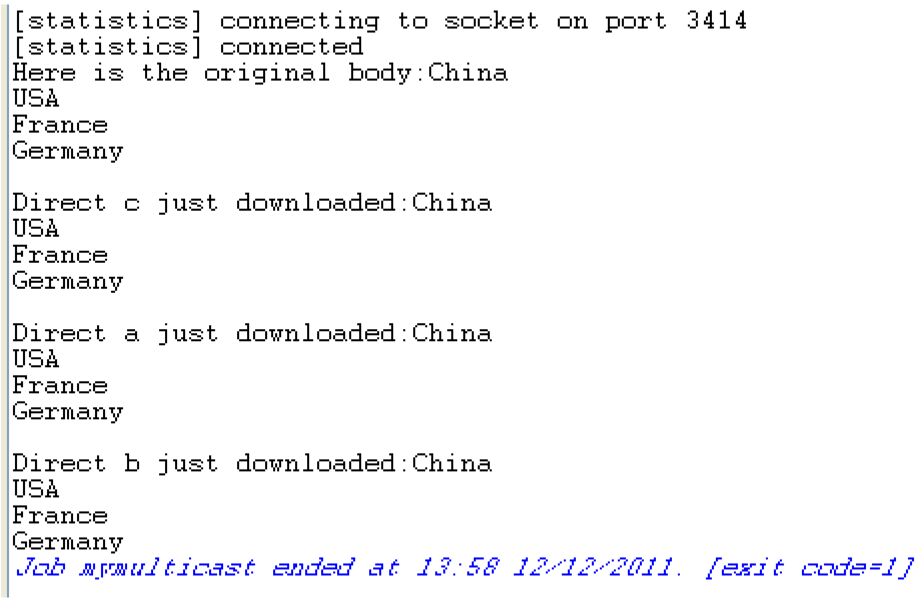


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("cFile_1")).routeId("cFile_1").process(
                    new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            String body = exchange.getIn().getBody(
                                    String.class);
                            System.out
                                    .println("Here is the original body:"
                                            + exchange
                                                    .getIn()
                                                    .getBody(
                                                            String.class));
                            ;
                        }

                    }).id("cProcessor_1").multicast().to(
                    "direct:a", "direct:b").id("cMulticast_1");
            from(uriMap.get("direct_a")).routeId("direct_a")
                    .enrich("direct:c")

                    .id("cContentEnricher_1").process(
                            new org.apache.camel.Processor() {
                                public void process(
                                        org.apache.camel.Exchange exchange)
                                        throws Exception {
                                    System.out
                                            .println("Direct a just downloaded:"
                                                    + exchange
                                                            .getIn()
                                                            .getBody(
                                                                    String.class));
                                    ;
                                }
```

As shown above, the route gets the original message from `cFile_1`, prints it out via `cProcessor_1`, and then `.multicast()` it `to( "direct:a", "direct:b")`. Afterwards, the message received by `direct_a` is used to `.enrich("direct:c")`.

2. Press **F6** to execute the Route.

   The original message is multicast to **direct_a** and **direct_b**. Also, it is used to enrich the contents received by **direct_c**.



```
[statistics] connecting to socket on port 3414
[statistics] connected
Here is the original body:China
USA
France
Germany

Direct c just downloaded:China
USA
France
Germany

Direct a just downloaded:China
USA
France
Germany

Direct b just downloaded:China
USA
France
Germany
Job mymulticast ended at 13:58 12/12/2011. [exit code=1]
```

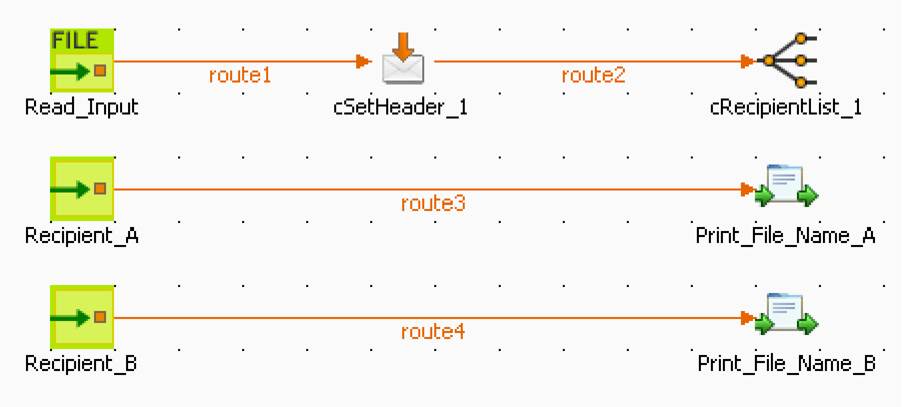
cRecipientList

cRecipientList properties

Component Family	Routing	
Function	**cRecipientList** is designed to route messages to a number of dynamically specified recipients.	
Purpose	**cRecipientList** allows you to route messages to a number of dynamically specified recipients.	
Basic settings	*Language*	Select the expression language from the drop-down list.
	Expression	Type in the expression that returns multiple endpoints.
	Stop On Exception	Select this check box to stop processing immediately when an exception occurred.
	Ignore Invalid Endpoints	Select this check box to ignore invalid endpoints.
	Parallel Processing	Select this check box to send the message to the recipients simultaneously.
Usage	As a middle component, **cRecipientList** allows you to route messages to a number of dynamically specified recipients.	
Limitation	n/a	

Scenario: Routing a message to multiple recipients

In this scenario, a **cRecipientList** component is used to route a message to a list of recipients.



To build the Route, do the following.

Dropping and linking the components

1. Drag and drop the components from the **Palette** onto the workspace: **cFile**, **cSetHeader**, **cRecipientList**, two **cMessagingEndpoint** and two **cProcessor**. Change the label of the **cFile** component to **Read_Input**. Change the labels of the two **cMessagingEndpoint** components to **Recipient_A** and **Recipient_B**. Change the labels of the two **cProcessor** components to **Print_File_Name_A** and **Print_File_Name_B**.

2. Link **Read_Input** to **cSetHeader** using a **Row** > **Route** connection.

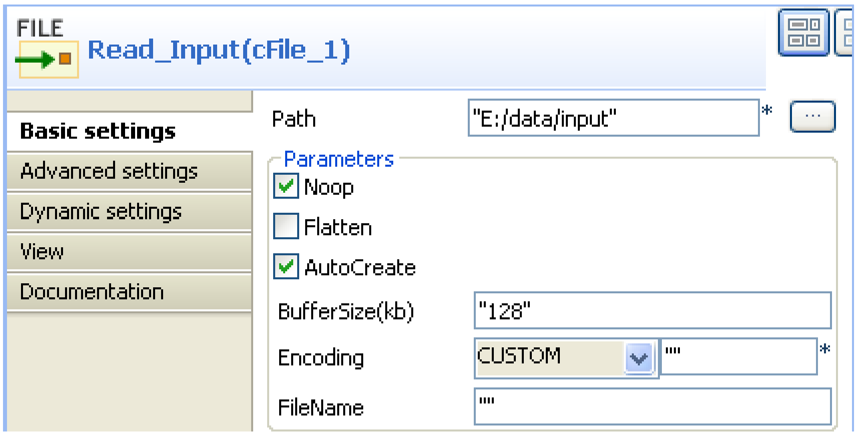
3. Link **cSetHeader** to **cRecipientList** using a **Row** > **Route** connection.

4. Link **Recipient_A** to **Print_File_Name_A** using a **Row** > **Route** connection.

5. Link **Recipient_B** to **Print_File_Name_B** using a **Row** > **Route** connection.

Configuring the components

1. Double-click **cFile** to open its **Basic settings** view in the **Component** tab.



Path "E:/data/input" *

☑ Noop

□ Flatten

☑ AutoCreate

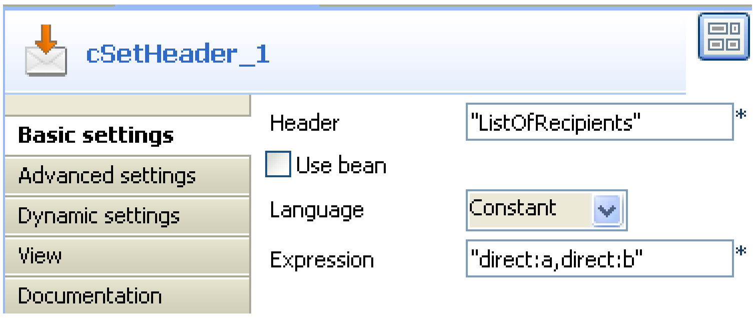
BufferSize(kb) "128"

Encoding CUSTOM "" *

FileName ""

2. In the **Path** field, type in the path to the source message, for example, *"E:/data/input"*. Keep other default settings unchanged.

3. Double-click **cSetHeader** to open its **Basic settings** view in the **Component** tab.



Header "ListOfRecipients" *

□ Use bean

Language Constant

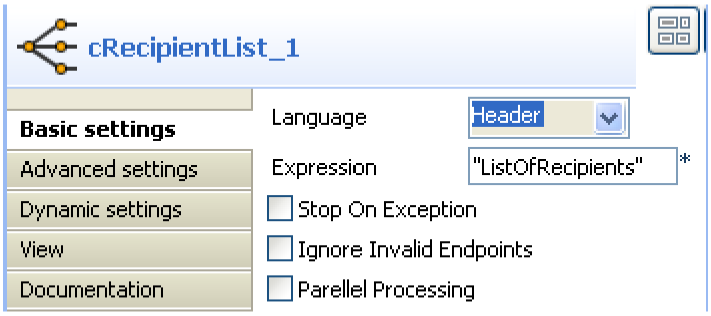
Expression "direct:a,direct:b" *

4. In the **Header** field, enter the header name, for example, *"ListOfRecipients"*.

   In the **Language** list, select *Constant*.

   In the **Expression** field, enter the endpoint URIs, for example, *"direct:a,direct:b"*.

5. Double-click **cRecipientList** to open its **Basic settings** view in the **Component** tab.



Language: Header

Expression: "ListOfRecipients"

□ Stop On Exception

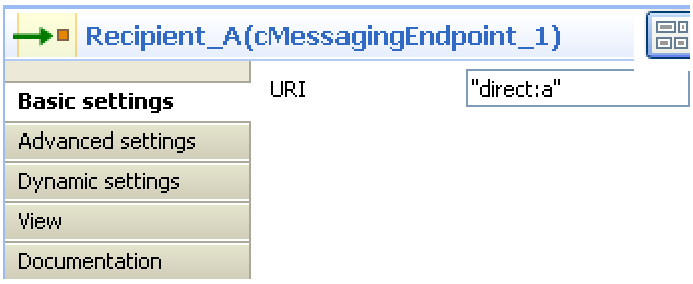
□ Ignore Invalid Endpoints

□ Parellel Processing

6. In the **Language** list, select **Header**.

   In the **Expression** field, enter the name of the header that contains the recipients list, that is, *"ListOfRecipients"*.

7. Double-click **Recipient_A** to open its **Basic settings** view in the **Component** tab and define the URI of recipient A.

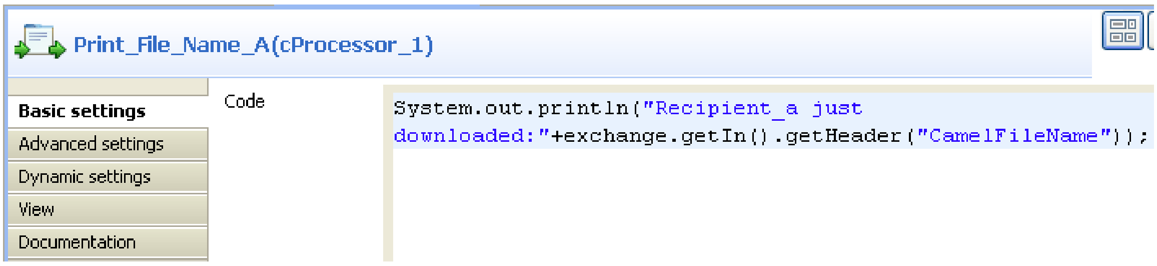


URI: "direct:a"

   Perform the same operation to **Recipient_B** to define the URI of recipient B.

8. Double-click **Print_File_Name_A** to open its **Basic settings** view in the **Component** tab and enter the code below to print out the message received by **Recipient_A**.

```
System.out.println("Recipient_a just
 downloaded:"+exchange.getIn().getHeader("CamelFileName"));
```



```
System.out.println("Recipient_a just
downloaded:"+exchange.getIn().getHeader("CamelFileName"));
```

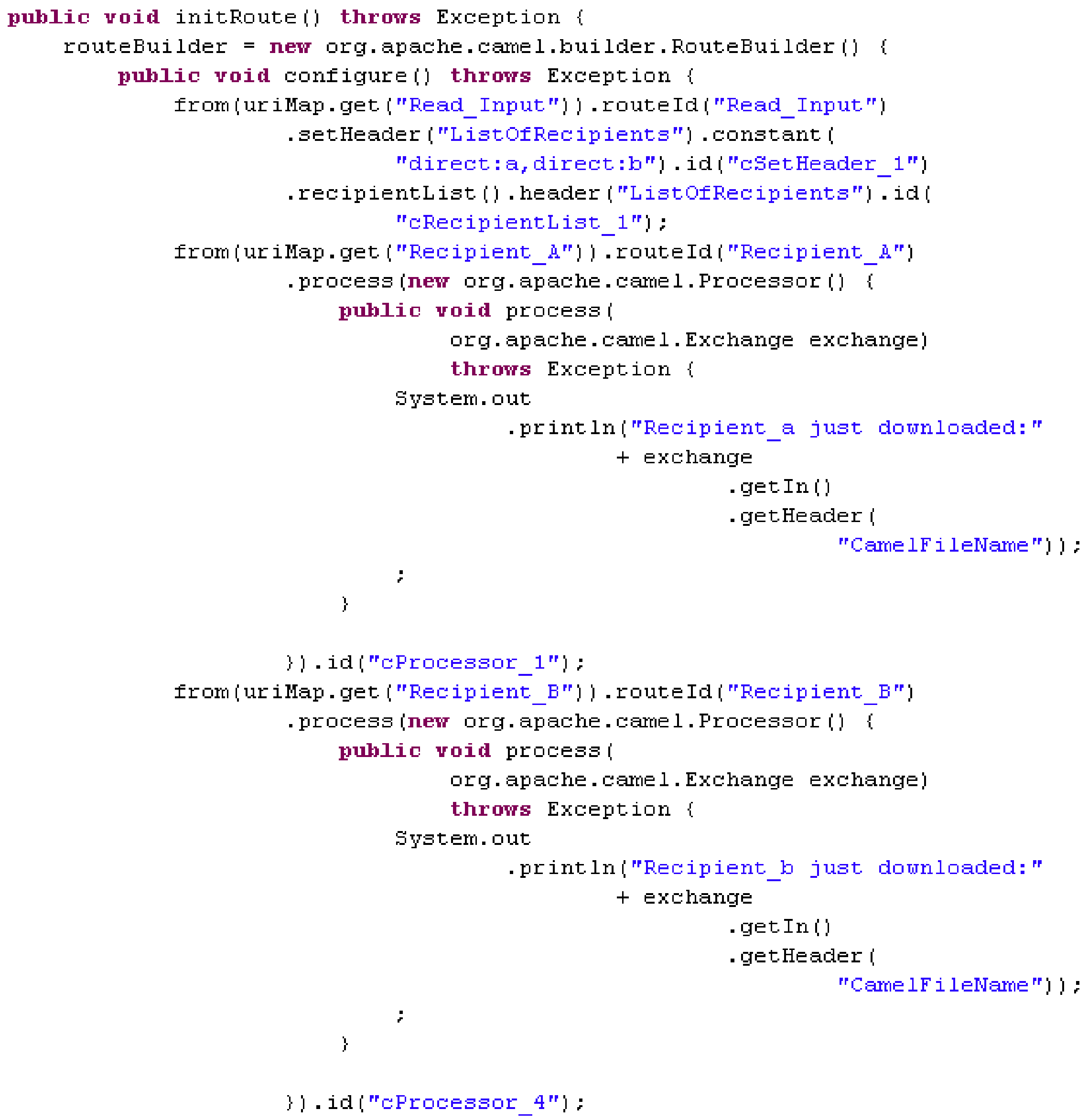
   Perform the same operation to **Print_File_Name_B** and type in the code below in its code box:

```
System.out.println("Recipient_b just
 downloaded:"+exchange.getIn().getHeader("CamelFileName"));
```

9. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.

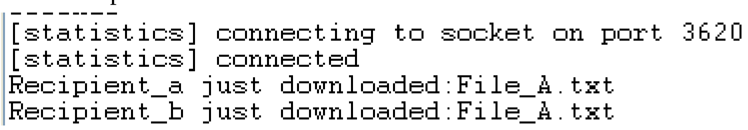


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Read_Input")).routeId("Read_Input")
                    .setHeader("ListOfRecipients").constant(
                            "direct:a,direct:b").id("cSetHeader_1")
                    .recipientList().header("ListOfRecipients").id(
                            "cRecipientList_1");
            from(uriMap.get("Recipient_A")).routeId("Recipient_A")
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out
                                    .println("Recipient_a just downloaded:"
                                            + exchange
                                                    .getIn()
                                                    .getHeader(
                                                            "CamelFileName"));
                            ;
                        }

                    }).id("cProcessor_1");
            from(uriMap.get("Recipient_B")).routeId("Recipient_B")
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out
                                    .println("Recipient_b just downloaded:"
                                            + exchange
                                                    .getIn()
                                                    .getHeader(
                                                            "CamelFileName"));
                            ;
                        }

                    }).id("cProcessor_4");
```

As shown above, the route gets the message from `Read_Input`, and `.setHeader("ListOfRecipients")` recipients using `.constant("direct:a,direct:b")`. Then, `cRecipientList_1` reads `.header("ListOfRecipients")` and routes the message to the recipients included in it.

2. Press **F6** to execute the Route.

The message is sent to recipients included in the header.



```
--------
[statistics] connecting to socket on port 3620
[statistics] connected
Recipient_a just downloaded:File_A.txt
Recipient_b just downloaded:File_A.txt
```

cSplitter

cSplitter properties

Component Family	Routing	
Function	**cSplitter** splits a message into several submessages according to a condition.	
Purpose	**cSplitter** separates multiple elements of a message so that they can be handled and treated differently in individual routes	
Basic settings	*Expression*	Type in the expression to use to split the messages.
Usage	**cSplitter** is used as a middle component in a Route.	
Connections	*split*	Select this link to route the splitted messages to the next endpoint.
	Route	Select this link to route all the messages from the sender to the next endpoint.
Limitation	n/a	

Related scenario:

For a related scenario, see the section called "Scenario: Splitting a message and renaming the sub-messages according to contained information" of the section called "cSetHeader".

cRoutingSlip

cRoutingSlip properties

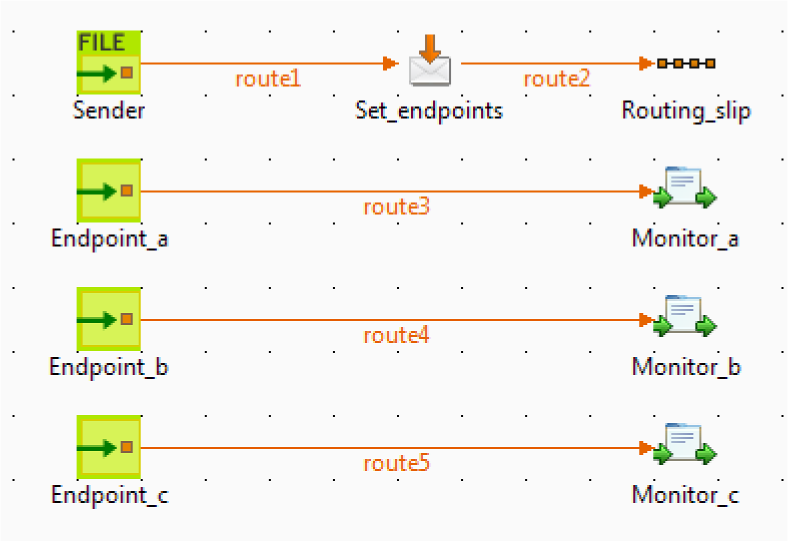
Component Family	Routing	
Function	**cRoutingSlip** allows you to route a message or messages consecutively through a series of processing steps, with the sequence of steps unknown at design time and variable for each message.	
Purpose	**cRoutingSlip** is used to route a message or messages consecutively to a series of endpoints.	
Basic settings	*Header name*	Type in name of the message header as defined in the preceding **cSetHeader** component, *mySlip* by default. The header should carry a list of endpoint URIs you wish each message to be routed to.
	URI delimiter	Delimiter used to separate multiple endpoint URIs carried in the message header, comma (,) by default.
Usage	**cRoutingSlip** is used as a middle or end component of a sub-route. It always follows a **cSetHeader** component, which sets a header to each message to carry a list of endpoint URIs.	
Limitation	n/a	

Scenario 1: Routing a message consecutively to a series of endpoints

In this scenario, messages from a file system is routed consecutively to a series of endpoints according to the URIs carried in the message header.

Dropping and linking the components

This use case requires a **cFile** component as the message sender, a **cSetHeader** component to define a series of endpoints, a **cRoutingSlip** component to route messages to the endpoints consecutively, three **cMessagingEndpoint** components to retrieve messages routed to the endpoints, and three **cProcessor** components to monitor messages routed to the connected messaging endpoints.



1. From the **Palette**, expand the **Messaging** folder, drop one **cFile** and three **cMessagingEndpoint** components onto the design workspace, and label them to better identify their roles in the Route, as shown above.

2. From the **Transformation** folder, drop a **cSetHeader** component onto the design workspace, and label it to better identify its role in the Route.

3. From the **Routing** folder, drop a **cRoutingSlip** component onto the design workspace, and label it to better identify its role in the Route.

4. From the **Processor** folder, drop three **cProcessor** components onto the design workspace, and label them to better identify their roles in the Route.

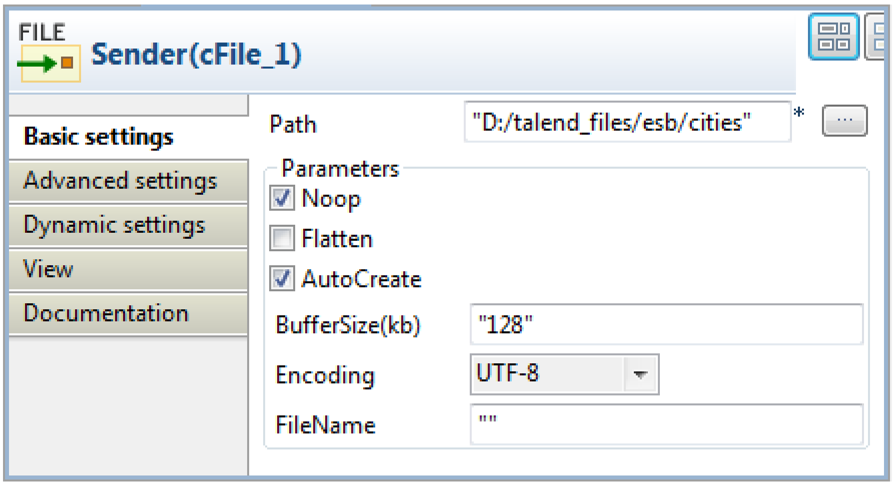
5. Right-click the **cFile** component, select **Row** > **Route** from the contextual menu and click the **cSetHeader** component.

6. Right-click the **cSetHeader** component, select **Row** > **Route** from the contextual menu and click the **cRoutingSlip** component.

7. Repeat this operation to connect the **cMessagingEndpoint** components to the corresponding **cProcessor** components.

Configuring the components and connections

1. Double-click the **cFile** component, which is labelled *Sender*, to display its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/esb/cities" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

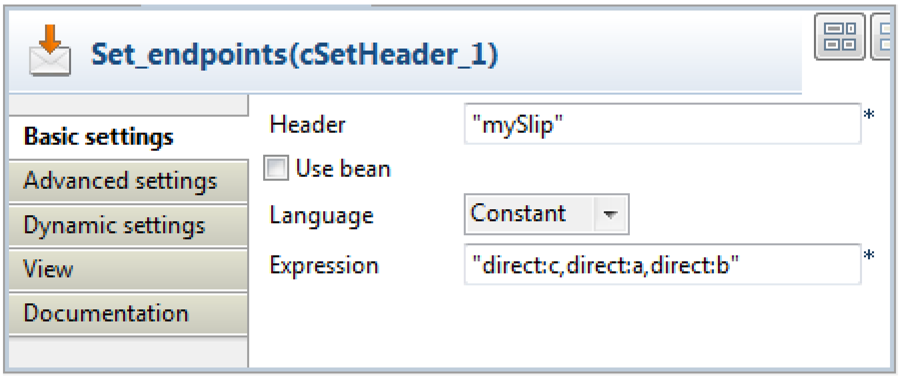
Encoding UTF-8

FileName ""

2. In the **Path** field, fill in or browse to the path to the folder that holds the source files.

   From the **Encoding** list, select the encoding type of your source files. Leave the other parameters as they are.

3. Double-click the **cSetHeader** component, which is labelled *Set_endpoints*, to display its **Basic settings** view in the **Component** tab.



Header "mySlip" *

□ Use bean

Language Constant

Expression "direct:c,direct:a,direct:b" *

4. In the **Header** field, type in the name of the header you want to add to each message.

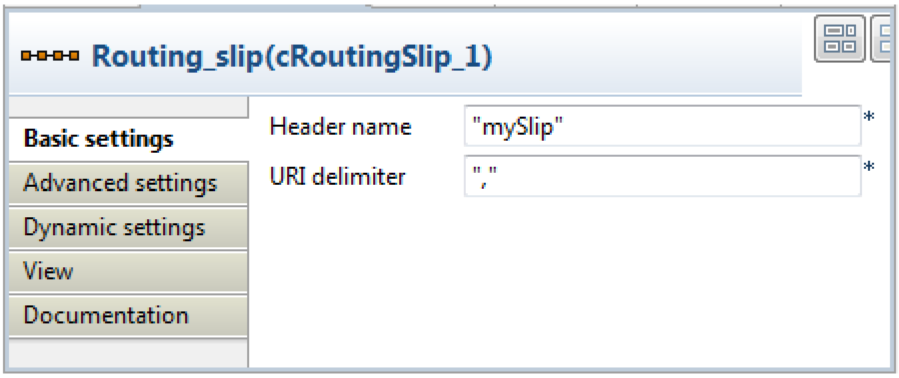
   In this use case, we simply use *mySlip*, which is the default value filled in the **Header name** field of the **cRoutingSlip** component.

5. From the **Language** list box, select the **Constant** or **Simple**, and in the **Expression** field, type in the URIs you wish the message to be routed consecutively to, separated by a comma, which is the default value of the **URI delimiter** field of the **cRoutingSlip** component.

   In this use case, we want the message to be routed first to endpoint *c*, then to endpoint *a*, and finally to endpoint *b*.

6. Double-click the **cRoutingSlip** component, which is labelled *Routing_slip*, to display its **Basic settings** view in the **Component** tab, and define the message header in the **Header name** field and the URI delimiter in the **URI delimiter** field.

   In this use case, we simply use the default settings.

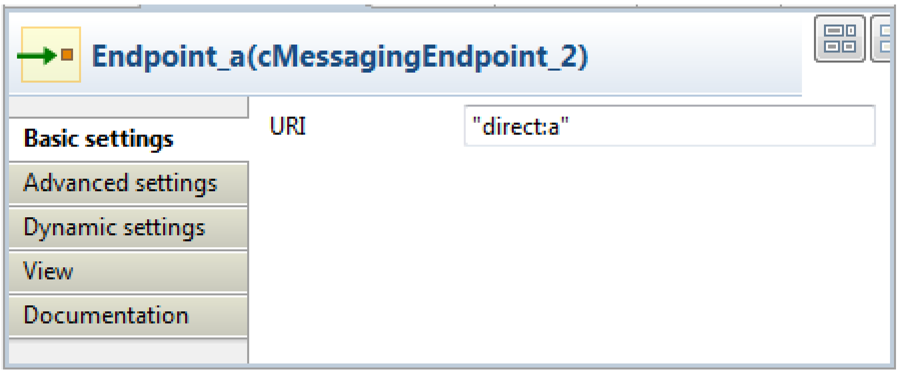


Header name "mySlip" *

URI delimiter "," *

7. Double-click the **cMessagingEndpoint** component labelled *Endpoint_a* to display its **Basic settings** view in the **Component** tab, and type in the URI in the **URI** field for the destination of your messages.

   Here, we want to use this component to retrieve the message routed to the URI *direct:a*.



Endpoint_a(cMessagingEndpoint_2)

URI "direct:a"

Repeat this step to set the endpoint URIs in the other **cMessagingEndpoint** components: *direct:b* and *direct:c* respectively.

8. Double-click the **cProcessor** component, which is labelled *Monitor_a*, to display its **Basic settings** view in the **Component** tab, and customize the code so that the console will display information the way you wish.

   Here, we want to use this component to monitor the messages routed to the connected endpoint *a* and display the file name, so we customize the code accordingly, as follows:

```
System.out.println("Message received on endpoint a: "+
exchange.getIn().getHeader("CamelFileName"));
```

   Repeat this step to customize the code for the other two **cProcessor** components, for messages routed to the connected endpoints *b* and *c* respectively.

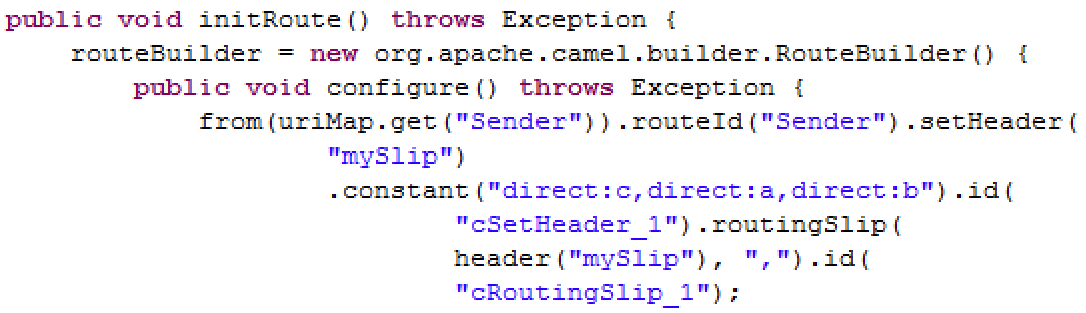
```
System.out.println("Message received on endpoint b: "+
exchange.getIn().getHeader("CamelFileName"));
```

```
System.out.println("Message received on endpoint c: "+
exchange.getIn().getHeader("CamelFileName"));
```

9. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

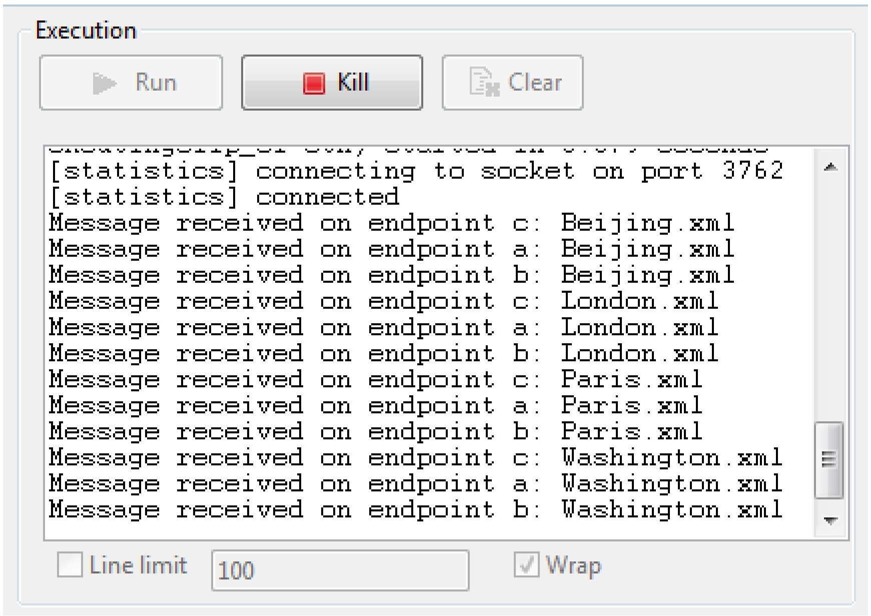


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender")).routeId("Sender").setHeader(
                    "mySlip")
                    .constant("direct:c,direct:a,direct:b").id(
                            "cSetHeader_1").routingSlip(
                            header("mySlip"), ",").id(
                            "cRoutingSlip_1");
```

   In this partially shown code, messages from the sender are given a header according to `.setHeader`, which carries a list of URIs (`"direct:c,direct:a,direct:b"`), and then routed in the slip pattern according by `cRoutingSlip_1`.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route.

   You can also press **F6** to execute it.



```
[statistics] connecting to socket on port 3762
[statistics] connected
Message received on endpoint c: Beijing.xml
Message received on endpoint a: Beijing.xml
Message received on endpoint b: Beijing.xml
Message received on endpoint c: London.xml
Message received on endpoint a: London.xml
Message received on endpoint b: London.xml
Message received on endpoint c: Paris.xml
Message received on endpoint a: Paris.xml
Message received on endpoint b: Paris.xml
Message received on endpoint c: Washington.xml
Message received on endpoint a: Washington.xml
Message received on endpoint b: Washington.xml
```

RESULT: The source file messages are routed consecutively to the defined endpoints: *c*, then *a*, and then *b*.

Scenario 2: Routing each message conditionally to a series of endpoints

In this scenario, which is based on the previous scenario, each message from a file system is routed consecutively to different endpoints according to the city name it contains.

All files used in this use case are named after the city name they contain. The following are the extracts of two examples:

Beijing.xml:

```
<person>
    <firstName>Nicolas</firstName>
    <lastName>Yang</lastName>
    <city>Beijing</city>
</person>
```

Paris.xml:

```
<person>
  <firstName>Pierre</firstName>
  <lastName>Dupont</lastName>
  <city>Paris</city>
</person>
```

A predefined Java Bean, *setEndpoints*, is called in this use case to return endpoint URIs according to the city name contained in each message, so that the messages will be routed as follows:

- The message containing the city name *Paris* will be routed first to endpoint *a*, then to endpoint *b*, and finally to endpoint *c*.

- The message containing the city name *Beijing* will be routed first to endpoint *c*, then to endpoint *a*, and finally to endpoint *b*.

- Any other messages will be routed to endpoint *b* and then to endpoint *c*.

For more information about creating and using Java Beans, see *Talend Open Studio for ESB* **User Guide**.

```
package beans;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class setEndpoints {
 public String helloExample(Document document) {
  NodeList cities = document.getDocumentElement().getElementsByTagName(
    "city");
  Element city = (Element) cities.item(0);
  String textContent = city.getTextContent();
  if ("Paris".equals(textContent)) {
   return "direct:a,direct:b,direct:c";
  } else if ("Beijing".equals(textContent)) {
   return "direct:c,direct:a,direct:b";
  } else
   return "direct:b,direct:c";
 }
}
```

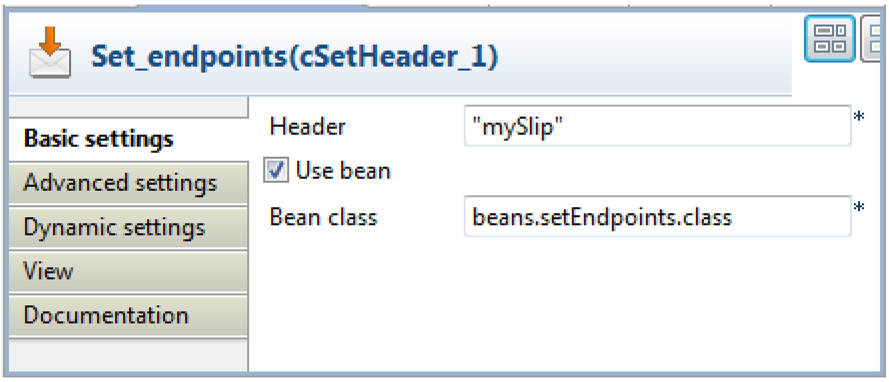
Dropping and linking the components

In this scenario, we will reuse the Route set up in the previous scenario, without adding or removing any components or modifying any connections.

Configuring the components and connections

In this scenario, we only need to configure the **cSetHeader** component to call the predefined Java Bean, and keep the settings of all the other components are they are in the previous scenario.

1. Double-click the **cSetHeader** component to display its **Basic settings** view in the **Component** tab.



Set_endpoints(cSetHeader_1)

Header "mySlip" *

☑ Use bean

Bean class beans.setEndpoints.class *

2. Select the **Use bean** check box, and in the **Bean class** field that appears, specify the Java Bean that will return the endpoint URIs. In this use case, type in

   `beans.setEndpoints.class`.

3. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

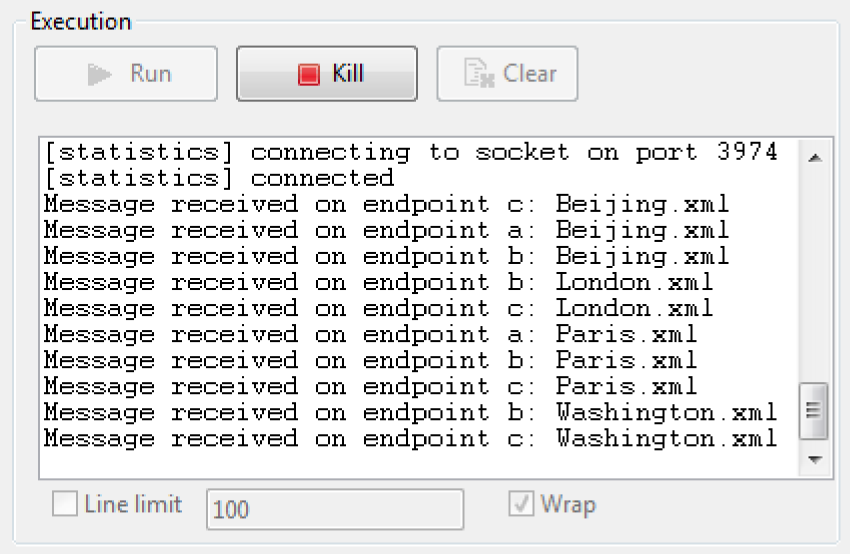
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender")).routeId("Sender").setHeader(
                    "mySlip").method(beans.setEndpoints.class).id(
                    "cSetHeader_1").routingSlip(header("mySlip"),
                    ",").id("cRoutingSlip_1");
```

   In this partially shown code, messages from the sender are given a header according to `.setHeader`, which carries a list of URIs returned by the `beans.setEndpoints.class`, and then routed to the `cRoutingSlip_1`.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route.

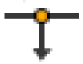
   You can also press **F6** to execute it.



```
[statistics] connecting to socket on port 3974
[statistics] connected
Message received on endpoint c: Beijing.xml
Message received on endpoint a: Beijing.xml
Message received on endpoint b: Beijing.xml
Message received on endpoint b: London.xml
Message received on endpoint c: London.xml
Message received on endpoint a: Paris.xml
Message received on endpoint b: Paris.xml
Message received on endpoint c: Paris.xml
Message received on endpoint b: Washington.xml
Message received on endpoint c: Washington.xml
```

   RESULT: The sources are routed consecutively to the defined endpoints: the message containing the city name *Beijing* is routed first to endpoint *c*, then to endpoint *a*, and finally to endpoint *b*; the message containing the city name *Paris* is routed first to endpoint *a*, then to endpoint *b*, and finally to endpoint *c*; the other messages are routed to endpoint *b* and then to endpoint *c*.

cWireTap

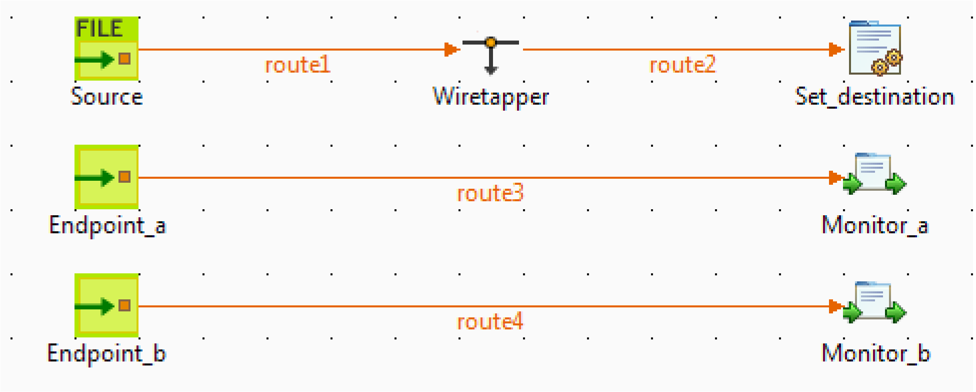


cWireTap properties

Component Family	Routing	
Function	**cWireTap** allows you to route messages to a separate tap endpoint while it is forwarded to the ultimate destination.	
Purpose	**cWireTap** is used to route messages to a separate endpoint while forwarded to the ultimate destination.	
Basic settings	*URI*	The endpoint URI to send the wire tapped message.
	Populate new exchange	Select this check box to populate a new exchange of the message.
	Populate Type	This option appears when the **Populate new exchange** check box is selected. The **Populate Type** is either **Expression** or **Processor**.
		Expression: Using expression allows you to set the message body of the new exchange. **Language**: Select the language of the expression you want to use to set the message body between **Constant**, **Header**, **None**, **Property**, **Simple**, **XPath**. **Expression**: Enter the expression to set the message body.
		Processor: Using processor gives you full power to specify how the exchange is populated as you can set properties, headers and so on to the message with a piece of Java code in the **Code** field.
	Copy the original message	Select this check box to use a copy of the exchange when wire tapping the message. This option appears when the **Populate new exchange** check box is selected.
Usage	**cWireTap** can be a middle component in a Route.	
Limitation	n/a	

Scenario: Wiretapping a message in a Route

In this scenario, a **cWireTap** component is used to route a message to a separate endpoint while it is routed to the ultimate destination.



Dropping and linking the components

1. From the **Palette**, expand the **Messaging** folder, and drop a **cFile** and two **cMessagingEndpoint** components onto the design workspace.

2. Expand the **Routing** folder, and drop a **cWireTap** component onto the design workspace.

3. Expand the **Processor** folder, and drop a **cJavaDSLProcessor** and two **cProcessor** components onto the design workspace.

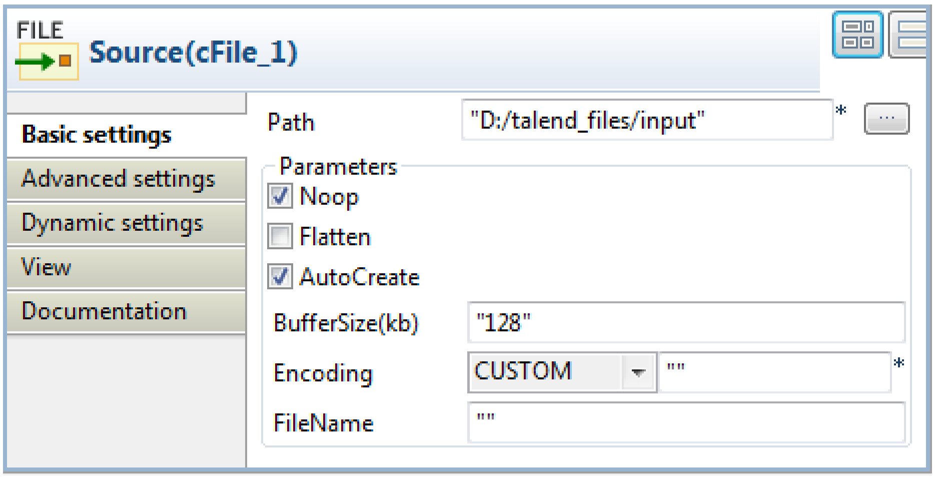
4. Right-click the **cFile** component, select **Row** > **Route** from the contextual menu and click the **cWireTap** component.

5. Repeat this operation to connect the components as shown above.

6. Label the components to better identify their functionality.

Configuring the components

1. Double-click the **cFile** component labeled *Source* to display its **Basic settings** view in the Component tab.


Source(cFile_1)

Path "D:/talend_files/input"

Parameters
☑ Noop
□ Flatten
☑ AutoCreate

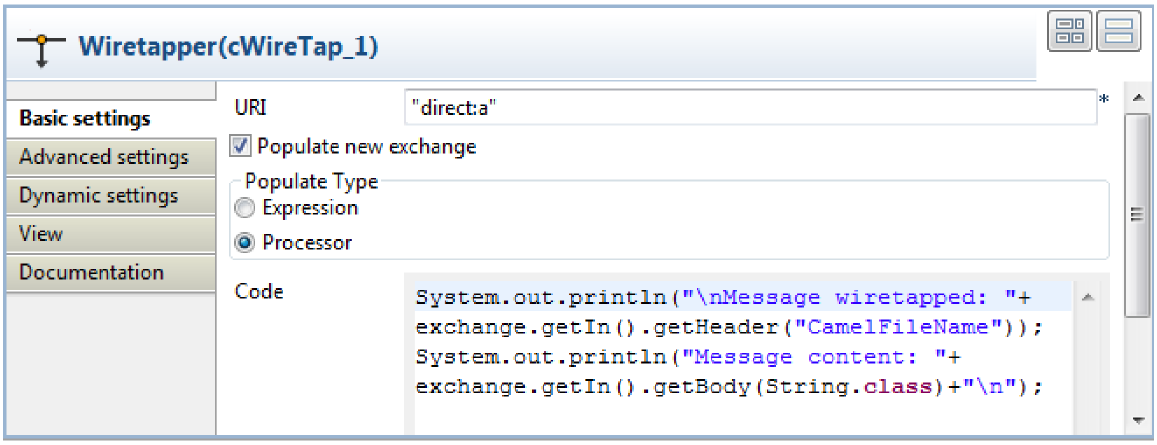
BufferSize(kb) "128"

Encoding CUSTOM ""

FileName ""

2. In the **Path** field, browse to or enter the input file path. In this use case, there is a *Hello.txt* file in the specified file path, which contains the content *Hello World!*. Leave the other parameters as they are.

3. Double-click the **cWireTap** component to display its **Basic settings** view in the **Component** tab.



URI "direct:a"

☑ Populate new exchange

Populate Type
○ Expression
● Processor

Code

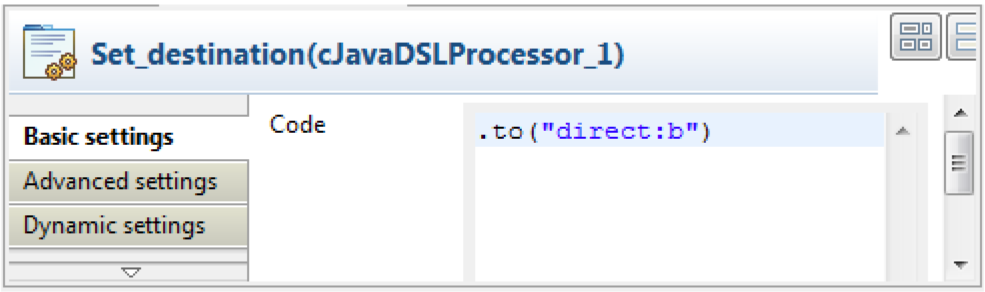
```
System.out.println("\nMessage wiretapped: "+
exchange.getIn().getHeader("CamelFileName"));
System.out.println("Message content: "+
exchange.getIn().getBody(String.class)+"\n");
```

4. Enter `"direct:a"` in the **URI** field to route the wiretapped message to this endpoint.

   Select the **Populate new exchange** check box, select **Processor** as the populate type, and then enter the following code in the **Code** box to display the file name of the wiretapped message and its content on the console:

```
System.out.println("\nMessage wiretapped: "+
exchange.getIn().getHeader("CamelFileName"));
System.out.println("Message content: "+
exchange.getIn().getBody(String.class)+"\n");
```

5. Double-click the **cJavaDSLProcessor** component to display its **Basic settings** view in the **Component** tab.

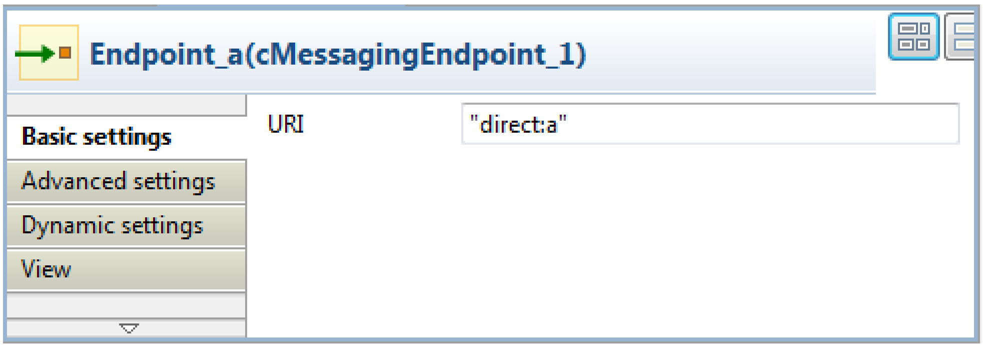


Code

```
.to("direct:b")
```

6. In the **Code** field, enter the Java code `.to("direct:b")` to define the URI of the endpoint to route the original message to.

7. Double-click the **cMessagingEndpoint** component labeled *Endpoint_a* to display its **Basic settings** view in the **Component** tab. Enter `"direct:a"` in the **URI** field to retrieve the message routed to this endpoint.

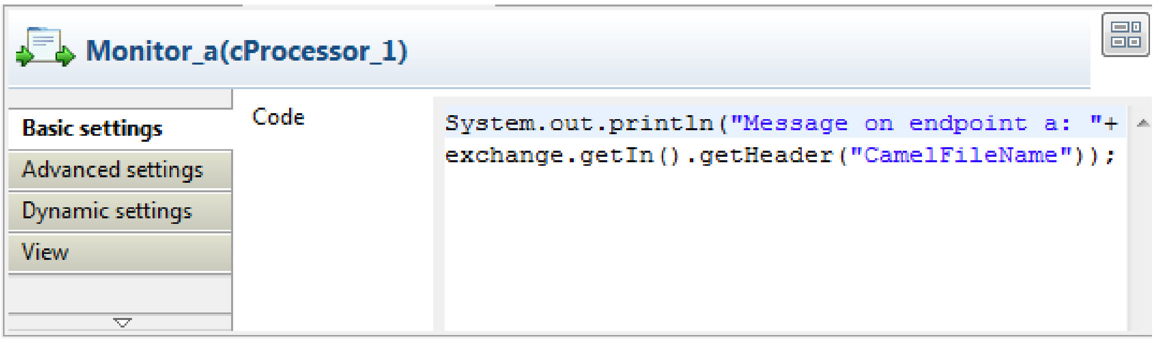


URI "direct:a"

   Repeat this operation to set the endpoint URI for *Endpoint_b*.

8. Double-click the **cProcessor** component labeled *Monitor_a* to display its **Basic settings** view in the **Component** tab. Enter the following code in the **Code** box to display the file name of the message routed to *Endpoint_a*.

```
System.out.println("Message on endpoint a: "+
exchange.getIn().getHeader("CamelFileName"));
```



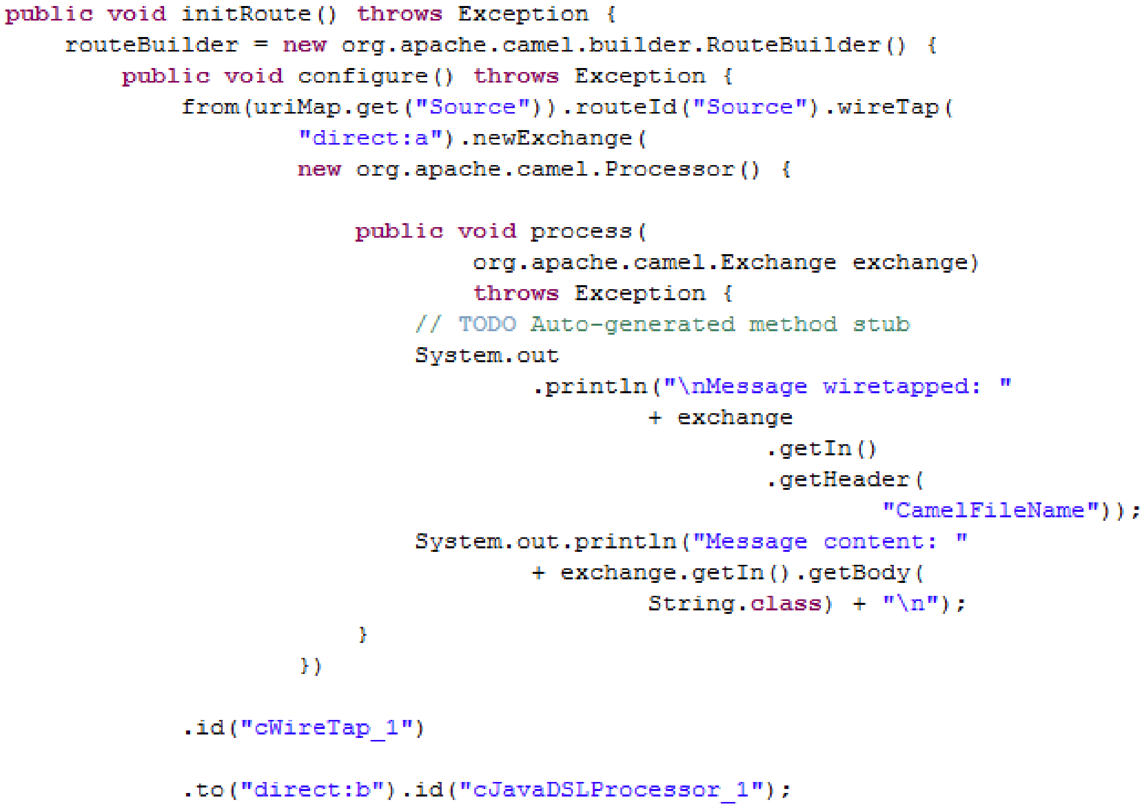
```
System.out.println("Message on endpoint a: "+
exchange.getIn().getHeader("CamelFileName"));
```

Then, configure the other **cProcessor** component in the same way to display the file name of the message routed to *Endpoint_b*.

9. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

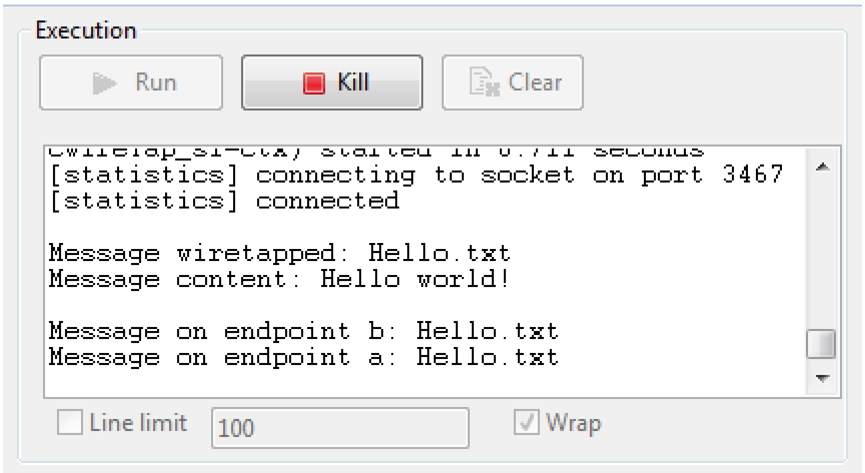


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Source")).routeId("Source").wireTap(
                    "direct:a").newExchange(
                    new org.apache.camel.Processor() {

                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            // TODO Auto-generated method stub
                            System.out
                                    .println("\nMessage wiretapped: "
                                            + exchange
                                                    .getIn()
                                                    .getHeader(
                                                            "CamelFileName"));
                            System.out.println("Message content: "
                                    + exchange.getIn().getBody(
                                            String.class) + "\n");
                        }
                    })

            .id("cWireTap_1")

            .to("direct:b").id("cJavaDSLProcessor_1");
```

In this partially shown code, any message `from` the endpoint `Source` will be wiretapped by `.wireTap` and routed to `"direct:a"`. The fine name and content of each wiretapped message will be displayed on the console. The original message will be routed `.to` an endpoint identified by the URI `"direct:b"`, which is defined in `cJavaDSLProcessor_1`.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route. You can also press **F6** to execute it.

RESULT: The source message is wiretapped and routed to endpoint *a* as well as being routed to endpoint *b*.



```
cWireTap_SI-ctx) started in 0.711 seconds
[statistics] connecting to socket on port 3467
[statistics] connected

Message wiretapped: Hello.txt
Message content: Hello world!

Message on endpoint b: Hello.txt
Message on endpoint a: Hello.txt
```

Transformation components

This chapter details the major components that you can find in **Transformation** family from the **Palette** of the **Mediation** perspective of *Talend Open Studio for ESB*.

The **Transformation** family groups component that execute data transformation processes.

cContentEnricher

cContentEnricher properties

Component Family	Transformation	
Function	**cContentEnricher** is designed to use a consumer or producer to obtain additional data, respectively intended for event messaging and request/reply messaging.	
Purpose	**cContentEnricher** allows you to use a consumer or producer to obtain additional data, respectively intended for event message messaging and request/reply messaging.	
Basic settings	*Resource URI*	This refers to the destination to which a message will be delivered if **using a producer** is selected; it refers to the source from which a message will be obtained if **using a consumer** is selected.
	Using a producer	Select this check box to use a producer to provide additional data, i.e. sending a message to the defined URI.
	Using a consumer	Select this check box to use a consumer to obtain additional data, i.e. requesting a message from the defined URI.
	Use Aggregation Strategy	Select this check box to define the aggregation strategy for assembling the basic message and the additional data.
	Specify timeout	This area appears when *Using a consumer* is selected. The timeout options are as follows: **Wait until a message arrive**: the component keeps waiting for a message. **Immediately polls the message**: the component immediately polls from the defined URI. **Waiting at most until the timeout triggers**: select this check box to type in a timeout value in Millis. The component waits for the message only within the defined time period.
Usage	**cContentEnricher** allows you to use a consumer or producer to obtain additional data, respectively intended for event message messaging and request/reply messaging.	
Limitation	n/a	

Related scenario

For a related scenario, see:

- **cMulticast**: the section called “Scenario: Multicasting a message to two endpoints and using it to enrich the contents received by the third endpoint”.

cConvertBodyTo

cConvertBodyTo properties

Component Family	Transformation	
Function	**cConvertBodyTo** converts the message body to the given class type.	
Purpose	**cConvertBodyTo** is used to convert the message body to a given class type.	
Basic settings	*Target Class Name*	Enter the name of the class type that you want to convert the message body to.
Usage	**cConvertBodyTo** is used as a middle component in a Route.	
Limitation		

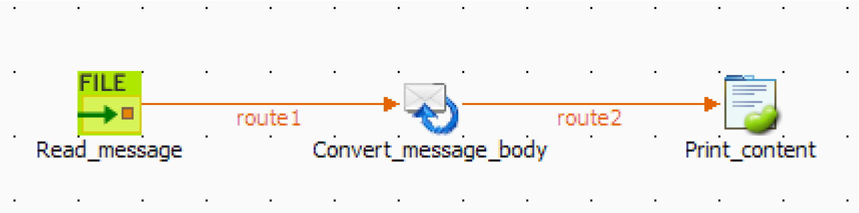
Scenario: Converting the body of an XML file into an org.w3c.dom.Document.class

In this scenario, a **cConvertBodyTo** component is used to convert the body of an XML file into an org.w3c.dom.Document.class. Then a **cBean** component imports the org.w3c.dom.Document class, checks its contents and prints out the root element name and the content of each **category** element.

The XML file is as follows:

```
<bookstore>
    <bookshelf>
        <category>Cooking</category>
        <quantity>100</quantity>
    </bookshelf>
    <bookshelf>
        <category>Languages</category>
        <quantity>200</quantity>
    </bookshelf>
    <bookshelf>
        <category>Arts</category>
        <quantity>300</quantity>
    </bookshelf>
    <bookshelf>
        <category>Science</category>
        <quantity>400</quantity>
    </bookshelf>
</bookstore>
```

Dropping and linking the components



1. Drag and drop the following components from the **Palette** onto the workspace: **cFile**, **cConvertBodyTo** and **cBean**.

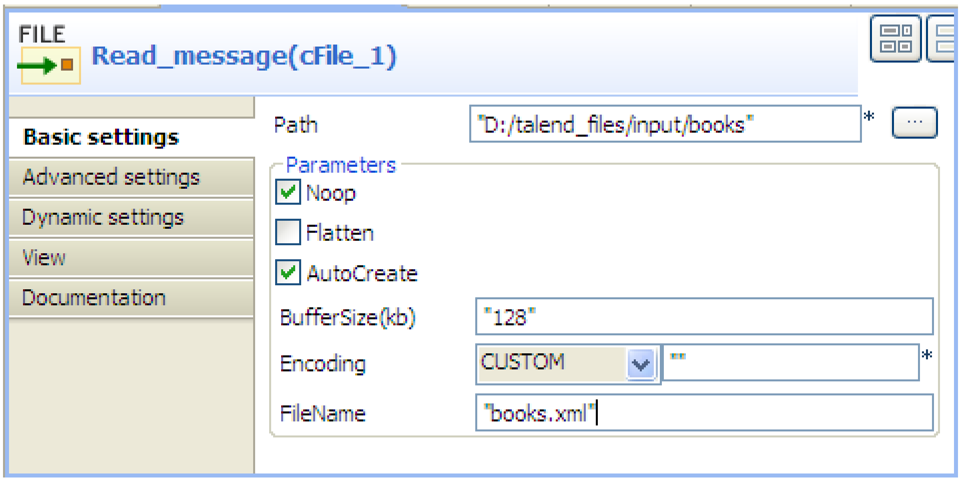
2. Link **cFile** to **cConvertBodyTo** using a **Row** > **Route** connection.

3. Link **cConvertBodyTo** to **cBean** using a **Row** > **Route** connection.

4. Label the components to better identify their functionality.

Configuring the components

1. Double-click the **cFile** component, which is labelled *Read_message*, to open its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/input/books" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

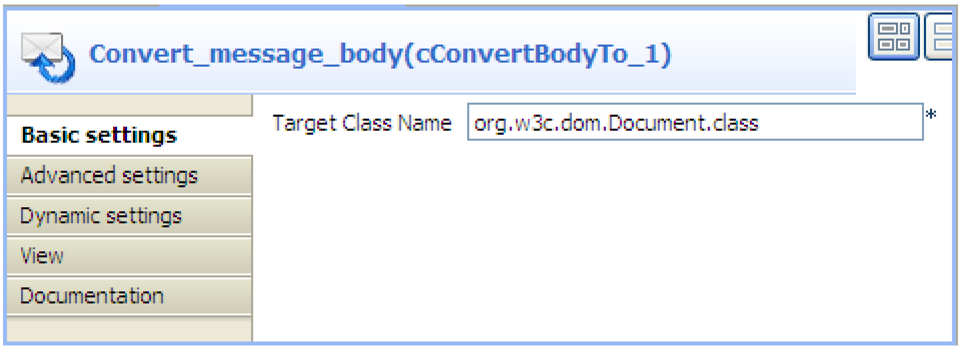
Encoding CUSTOM "" *

FileName "books.xml"

2. In the **Path** field, enter or browse to the path to the source XML file.

3. If the source file folder contains more than one file, enter the name of the XML file of interest in the **FileName** field, and leave the other parameters as they are.

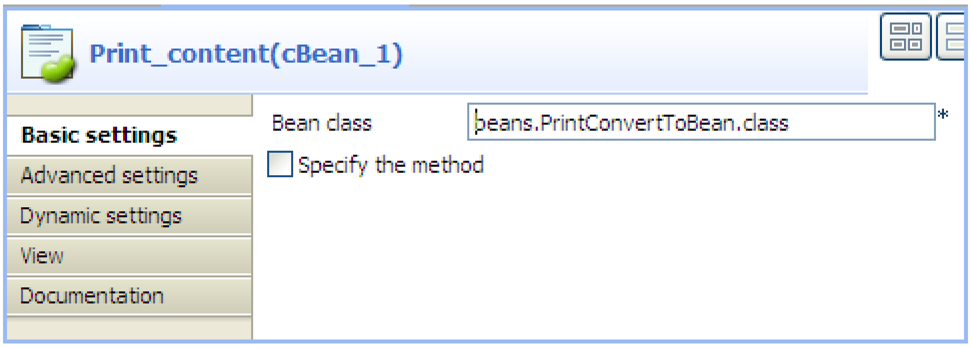
4. Double-click the **cConvertBodyTo** component, which is labelled *Convert_message_body*, to open its **Basic settings** view in the **Component** tab.



Target Class Name org.w3c.dom.Document.class *

5. In the **Target Class Name** field, enter your target class name, *org.w3c.dom.Document.class* in this scenario.

6. Double-click the **cBean** component, which is labelled *Print_content*, to open its **Basic settings** view in the **Component** tab.



Bean class: beans.PrintConvertToBean.class

☐ Specify the method

7. In the **Bean class** field, enter the name of the bean to be invoked, *beans.PrintConvertToBean.class* in this scenario.

   Note that this bean has already been defined in the **Code** node of the **Repository** and it looks like this:

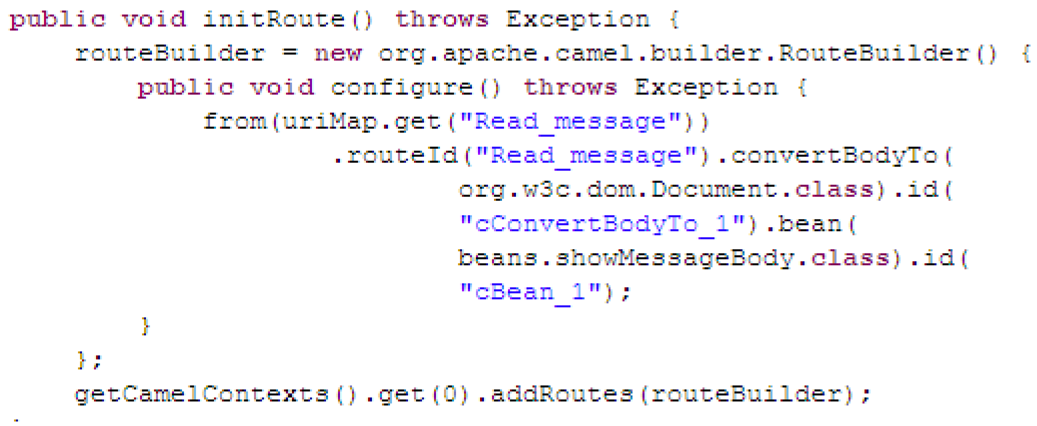
```
package beans;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
public class PrintConvertToBean {

 /**
  * print input message
  * @param message
  */
 public static void helloExample(Document message) {
  if (message == null) {
   System.out.println("There's no message here!");
   return;
  }
  Element rootElement = message.getDocumentElement();
  if (rootElement == null) {
   System.out.println("There's no root element here!");
   return;
  }
  System.out.println("The root element name is:"
    + rootElement.getNodeName());
  System.out.println("The book categories are:");
  NodeList types = rootElement.getElementsByTagName("category");
  for(int i = 0;i<types.getLength();i++){
   Element child = (Element) types.item(i);
   System.out.println(child.getFirstChild().getNodeValue());
  }
 }
}
```

   For more information about creating and using Java Beans, see *Talend Open Studio for ESB* **User Guide**.

8. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.

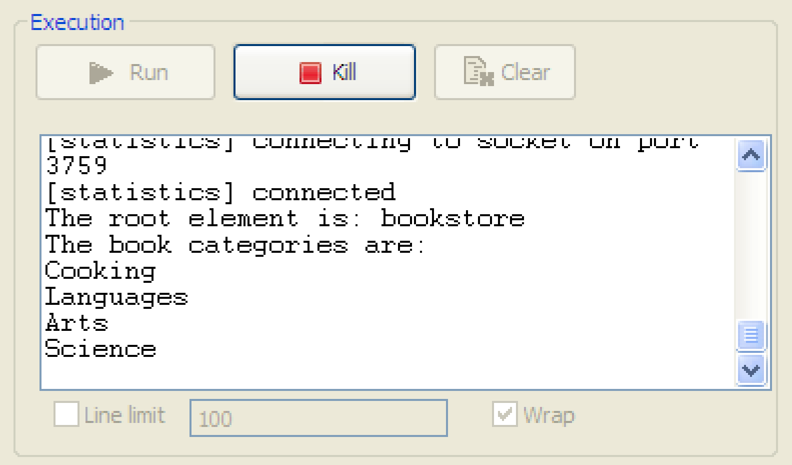


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Read_message"))
                    .routeId("Read_message").convertBodyTo(
                            org.w3c.dom.Document.class).id(
                            "cConvertBodyTo_1").bean(
                            beans.showMessageBody.class).id(
                            "cBean_1");
        }
    };
    getCamelContexts().get(0).addRoutes(routeBuilder);
```

As shown above, the message `from` the endpoint `Read_message` has its body converted to `org.w3c.dom.Document.class` by `cConvertBodyTo_1`. Then, `org.w3c.dom.Document.class` is processed by `.bean(beans.PrintConvertToBean.class)` invoked by `cBean_1`.

2. Press **F6** to execute the Route.

RESULT: The root element name and the contents of the **category** elements are displayed.



```
[statistics] connecting to socket on port
3759
[statistics] connected
The root element is: bookstore
The book categories are:
Cooking
Languages
Arts
Science
```

cSetBody

cSetBody properties

Component Family	Transformation	
Function	**cSetBody** replaces the payload of each message sent to it.	
Purpose	**cSetBody** is used to replace the content of each message sent to it according to expression value.	
Basic settings	*Language*	Select the language of the expression you use to set the content for matched messages, from **Constant**, **EL**, **Groovy**, **Header**, **JavaScript**, **JoSQL**, **JXPath**, **MVEL**, **None**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**.
	Expression	Type in the expression to set the message content.
Usage	**cSetBody** is used as a middle component in a Route.	
Limitation	n/a	

Scenario: Replacing the content of messages with their extracts

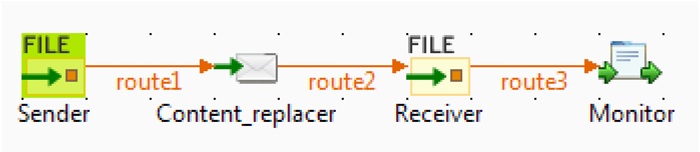
In this scenario, file messages are routed from one endpoint to another, with the content of each message replaced with the information extracted from it.

The following is an example of the XML files used in this use case:

```
<people>
    <person>
        <firstName>Pierre</firstName>
        <lastName>Dubois</lastName>
        <city>Paris</city>
    </person>
</people>
```

Dropping and linking the components

This use case uses two **cFile** components, one as the message sender and the other as the receiver, a **cSetBody** component to replace the content of the messages on route, and a **cProcessor** component to display the new content of the messages routed to the receiving endpoint.



1. From the **Palette**, expand the **Messaging** folder, and drop two **cFile** components onto the design workspace.

2. From the **Transformation** folder, drop a **cSetBody** component onto the design workspace, between the two **cFile** components.

3. From the **Processor** folder, drop a **cProcessor** component onto the design workspace, following the second **cFile** component.

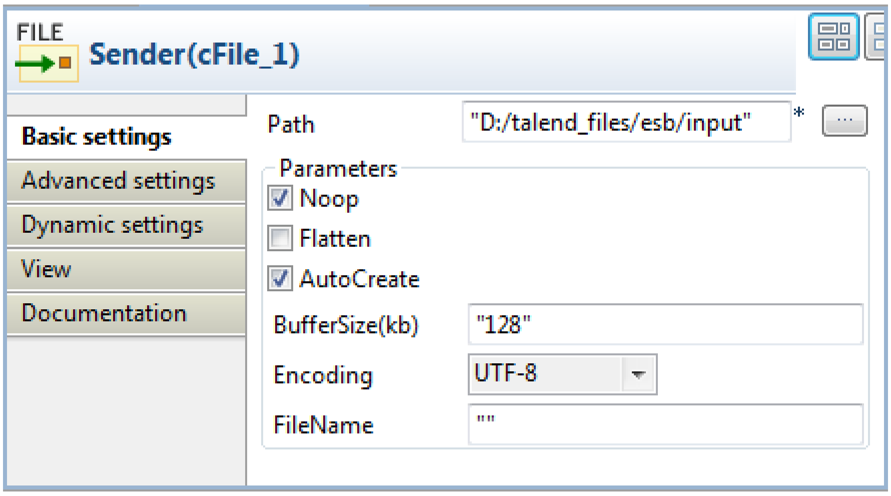
4. Right-click the first **cFile** select **Row** > **Route** from the contextual menu and click the **cSetBody** component.

5. Repeat this operation to connect the **cSetBody** component to the second **cFile** component, and the second **cFile** component to the **cProcessor** component.

6. Label the components to better identify their roles in the Route, as shown above.

Configuring the components and connections

1. Double-click the **cFile** component labeled *Sender* to display its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/esb/input" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

BufferSize(kb) "128"

Encoding UTF-8

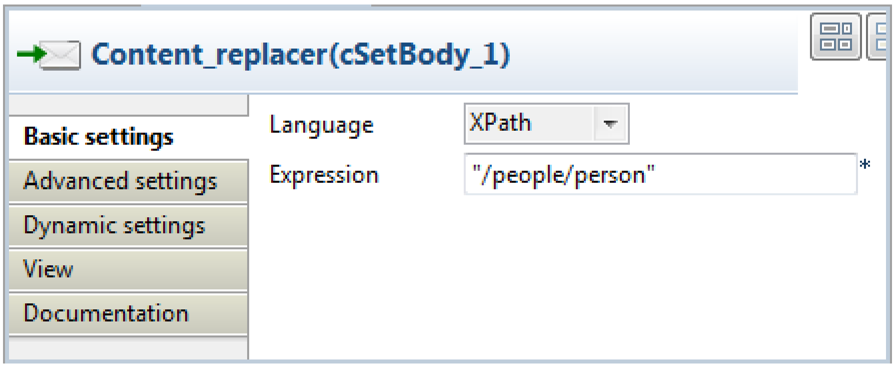
FileName ""

2. In the **Path** field, fill in or browse to the path to the folder that holds the source files.

3. From the **Encoding** list, select the encoding type of your source files. Leave the other parameters as they are.

4. Repeat these steps to define output file path and encoding type in the **Basic settings** view of the other **cFile** component, which is labeled *Receiver*.

5. Double-click the **cSetBody** component to display its **Basic settings** view in the **Component** tab.



Language XPath

Expression "/people/person" *

6. From the **Language** list box, select the language of the expression you are going to use.

   Here we are handling XML files, so select **XPath** from the list box.

7. In the **Expression** field, type in the expression that will return the new message content you want.

   In this use case, we want *person* to be the root element of each file when routed to the receiving endpoint, so type in `"/people/person"` in the **Expression** field.

8. Double-click the **cProcessor** component to display its **Basic settings** view in the **Component** tab, and customize the code so that the console will display information the way you wish.

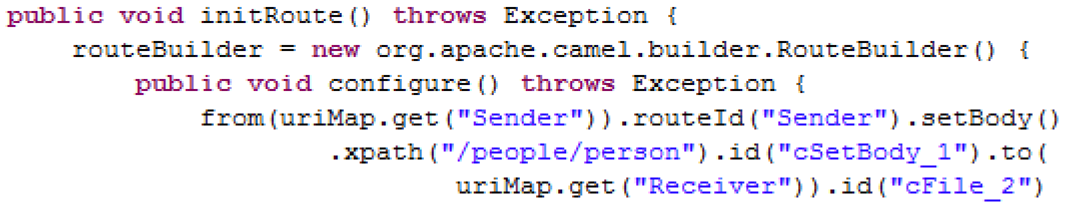
   In this use case, we want to display the file name and content of each message routed to the receiving endpoint, so we customize the code as follows:

```
System.out.println("File received: " +
exchange.getIn().getHeader("CamelFileName") +
"\nContent:\n " +
exchange.getIn().getBody(String.class));
```

9. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

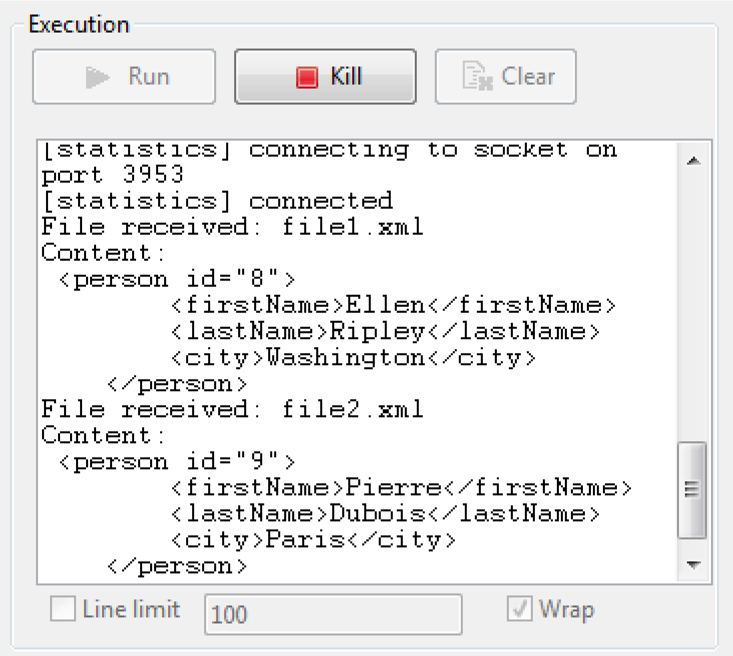
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.



```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender")).routeId("Sender").setBody()
                    .xpath("/people/person").id("cSetBody_1").to(
                            uriMap.get("Receiver")).id("cFile_2")
```

   In this partially shown code, a message route is built `from` one endpoint `.to` another, and while in routing, the content of each message is replaced according to the condition `.xpath("/people/person")` by `"cSetBody_1"`.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route. You can also press **F6** to execute it.



```
[statistics] connecting to socket on
port 3953
[statistics] connected
File received: file1.xml
Content:
 <person id="8">
        <firstName>Ellen</firstName>
        <lastName>Ripley</lastName>
        <city>Washington</city>
    </person>
File received: file2.xml
Content:
 <person id="9">
        <firstName>Pierre</firstName>
        <lastName>Dubois</lastName>
        <city>Paris</city>
    </person>
```

RESULT: The XML files are sent to the receiver, where *person* has become the root element of each file.

cSetHeader

cSetHeader properties

Component Family	Transformation	
Function	**cSetHeader** sets a header on each message sent to it.	
Purpose	**cSetHeader** is used to set a header or customize the default header, if any, on each message sent to it for subsequent message processing.	
Basic settings	*Header*	Type in a name for the message header.
	Use bean	Select this check box if you want to call a predefined Java Bean to return the header value. When this check box is selected, a **Bean class** field appears for you specify the Bean class to call.
	Bean class	Type in the Bean class that will return a value for the message header, in the form of *beans.BEAN_NAME.class*.
	Language	Select the language of the expression you use, from **Constant**, **EL**, **Groovy**, **Header**, **JavaScript**, **JoSQL**, **JXPath**, **MVEL**, **None**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**. This list box is hidden when the **Use bean** check box is selected.
	Expression	Type in the expression to set the value of the message header. This field is hidden when the **Use bean** check box is selected.
Usage	**cSetHeader** is used as a middle component in a Route.	
Limitation	n/a	

Scenario: Splitting a message and renaming the sub-messages according to contained information

In this scenario, a file message containing people information is split into sub-messages. Each sub-messages is renamed according the city name it contains, and then routed to another endpoint.

The following is the example XML file used in this use case:

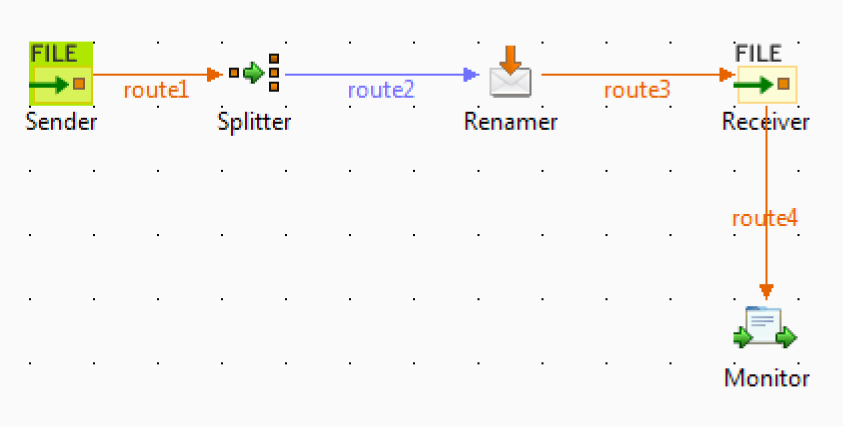
```
<people>
    <person>
        <firstName>Pierre</firstName>
        <lastName>Dubois</lastName>
        <city>Paris</city>
    </person>
    <person>
        <firstName>Nicolas</firstName>
        <lastName>Yang</lastName>
        <city>Beijing</city>
    </person>
    <person>
        <firstName>Ellen</firstName>
        <lastName>Ripley</lastName>
        <city>Washington</city>
    </person>
</people>
```

A predefined Java Bean, *setFileNames*, is called by the **cSetHeader** component used in this use case to define a file name for each message according to the city name it contains. For more information about creating and using Java Beans, see *Talend Open Studio for ESB* **User Guide**.

```
package beans;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class setFileNames {
 public String getCityName(Document document) {
  NodeList cities = document.getDocumentElement().getElementsByTagName(
    "city");
  Element city = (Element) cities.item(0);
  String textContent = city.getTextContent();
   return textContent+".xml";
  }
}
```

Dropping and linking the components

This use case uses two **cFile** components, one as the message sender and the other as the receiver, a **cSplitter** component to split the source message into sub-messages, a **cSetHeader** component to rename each sub-message, and a **cProcessor** component to display the file name of each message routed to the receiver.



1. From the **Palette**, expand the **Messaging** folder, and drop two **cFile** components onto the design workspace.

2. From the **Routing** folder, drop a **cSplitter** component onto the design workspace, between the two **cFile** components.

3. From the **Transformation** folder, drop a **cSetHeader** component onto the design workspace, between the **cSplitter** component and the receiving **cFile** component.

4. Right-click the first **cFile** component, select **Row** > **Route** from the contextual menu and click the **cSplitter** component.

5. Right-click the **cSplitter** component, select **Row** > **Split** from the contextual menu and click the **cSetHeader** component.

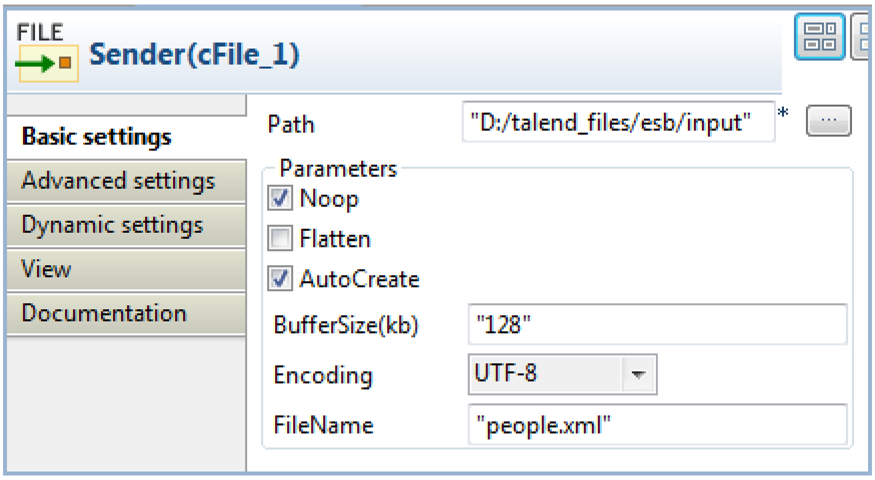
6. Right-click the **cSetHeader** component, select **Row** > **Route** from the contextual menu and click the second **cFile** component.

7. Right-click the second **cFile** component, select **Row** > **Route** from the contextual menu and click the **cProcessor** component.

8. Label the components to better identify their roles in the Route, as shown above.

Configuring the components and connections

1. Double-click the **cFile** component labeled *Sender* to display its **Basic settings** view in the **Component** tab.



Path "D:/talend_files/esb/input" *

Parameters

☑ Noop

☐ Flatten

☑ AutoCreate

BufferSize(kb) "128"

Encoding UTF-8

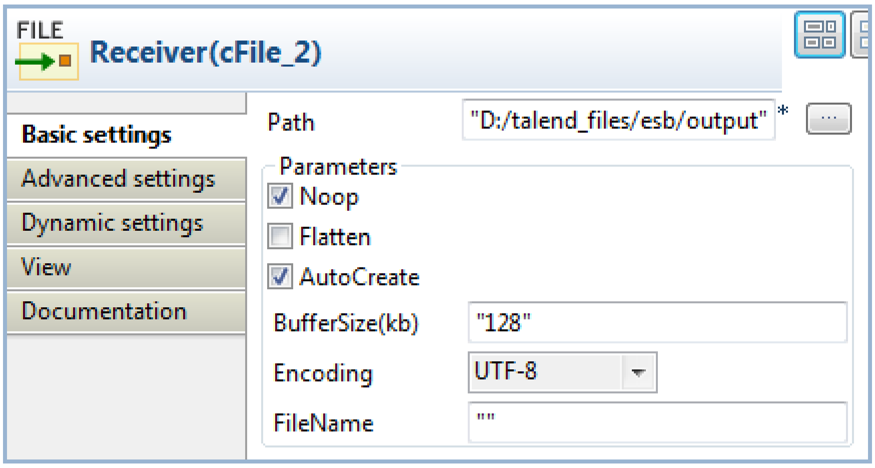
FileName "people.xml"

2. In the **Path** field, fill in or browse to the path to the folder that holds the source files.

   From the **Encoding** list, select the encoding type of your source files.

   In the **FileName** field, type in the file name of the source message. You can skip this step if the source folder contains only one file.

3. Repeat steps 1 and 2 above to define the output file path and encoding type in the **Basic settings** view of the other **cFile** component, which is labeled *Receiver*. Leave the **FileName** field blank.



Receiver(cFile_2)

Path "D:/talend_files/esb/output" *

Parameters

☑ Noop

□ Flatten

☑ AutoCreate

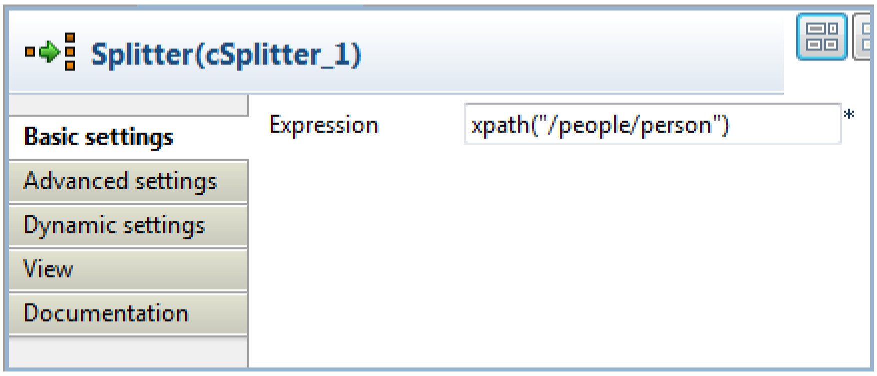
BufferSize(kb) "128"

Encoding UTF-8

FileName ""

4. Double-click the **cSplitter** component to display its **Basic settings** view in the **Component** tab, and fill the **Expression** field with an expression according to which you want to split the source message.

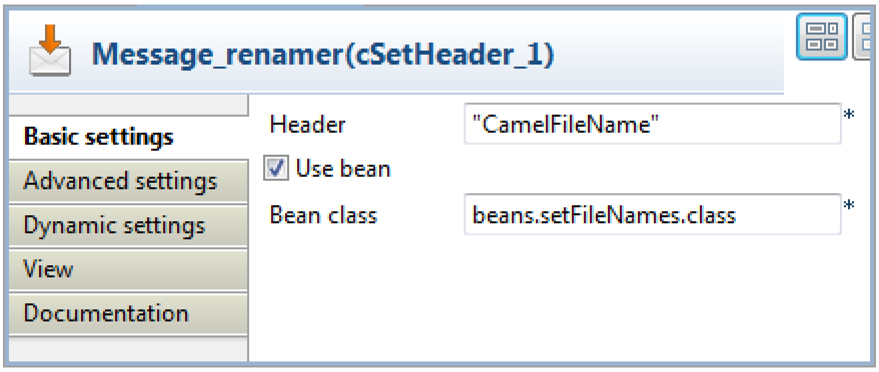
   In this use, as we want to split the message into sub-messages at each *person* node of the XML file, type in `xpath("/people/person")`.



Splitter(cSplitter_1)

Expression xpath("/people/person") *

5. Double-click the **cSetHeader** component, which is labeled *Message_renamer* to display its **Basic settings** view in the **Component** tab.



Message_renamer(cSetHeader_1)

Header "CamelFileName" *

☑ Use bean

Bean class beans.setFileNames.class *

6. In the **Header** field, type in the name of the header you want to give to the messages.

   Here, as we want to define the file name for each incoming message, fill in `"CamelFileName"` as the header name.

7. Select the **Use bean** check box, and in the **Bean class** field that appears, type in the name of the predefined Java Bean. In this use case, type in `beans.setFileNames.class`.

8. Double-click the **cProcessor** component to display its **Basic settings** view in the **Component** tab, and customize the code so that the console will display information the way you wish.

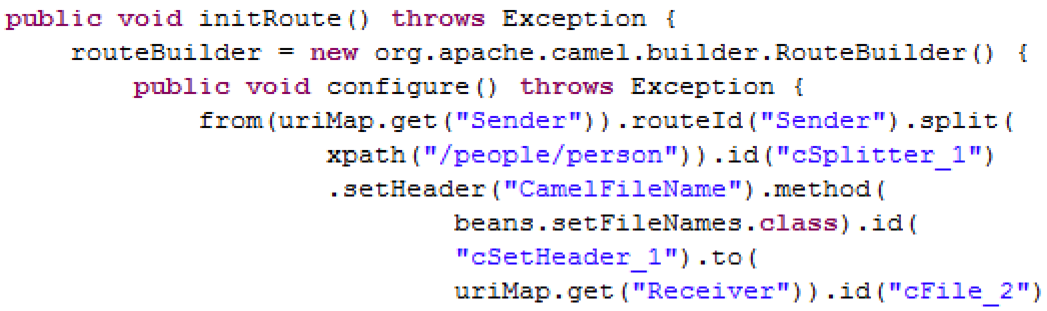
In this use case, we want to display the file name each message routed to the receiving endpoint, so we customize the code as follows:

```
System.out.println("File received: "+
exchange.getIn().getHeader("CamelFileName"));
```

9. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

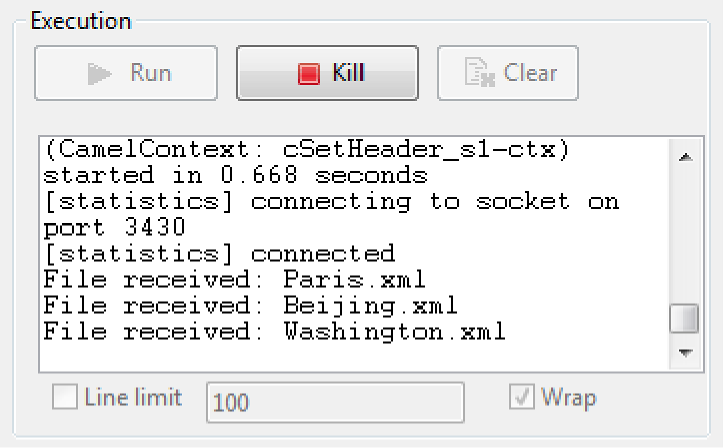
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.



```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Sender")).routeId("Sender").split(
                    xpath("/people/person")).id("cSplitter_1")
                    .setHeader("CamelFileName").method(
                            beans.setFileNames.class).id(
                            "cSetHeader_1").to(
                            uriMap.get("Receiver")).id("cFile_2")
```

As shown in the code, a message route is built `from` one endpoint `.to` another, and while in routing, the source message is split according to the condition `xpath("/people/person")` by `cSplitter_1`, and each sub-message is given a header named `CamelFileName`, the value of which is returned by `.method(beans.setFileName.class)`.

2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route. You can also press **F6** to execute it.



```
(CamelContext: cSetHeader_s1-ctx)
started in 0.668 seconds
[statistics] connecting to socket on
port 3430
[statistics] connected
File received: Paris.xml
File received: Beijing.xml
File received: Washington.xml
```

RESULT: The source file message is split into sub-messages and each sub-message is renamed after the city name it contains and routed to the receiving endpoint.

Related scenarios

For more scenarios, see:

the section called “Scenario: Using camel message headers as context parameters to call a job” the section called “Scenario 1: Routing a message consecutively to a series of endpoints”

the section called “Scenario 2: Routing each message conditionally to a series of endpoints”