talend | Connecting the Data-Driven Enterprise

Talend Mediation Components Reference Guide

6.4.1

Contents

Copyright

cAggregate properties

cAggregate allows you to combine a number of messages together into a single message.

cAggregate aggregates messages together according to specified conditions.

cAggregate Standard properties

These properties are used to configure cAggregate running in the Standard Job framework.

The Standard cAggregate component belongs to the Routing family.

Basic settings

Language	Select the language of the expression you want to use to filter your messages, from **None**, **Bean**, **CONSTANT**, **ESB[CorrelationID]**, **EL**, **GROOVY**, **HEADER**, **JAVASCRIPT**, **JoSQL**, **JSonPath**, **JXPATH**, **MVEL**, **OGNL**, **PHP**, **PROPERTY**, **PYTHON**, **RUBY**, **SIMPLE**, **SpEL**, **SQL**, **XPATH**, and **XQUERY**. Select **CorrelationID** to use the existing correlation ID of the message as the correlation key if the correlation ID is available in the closest **cSOAP** connected to this component. For more information about the **cSOAP** component, see cSOAP properties on page 53.
Correlation expression/Expression	Type in the expression that evaluates the correlation key to be used for the aggregation. This field disappears when **CorrelationID** is selected in the **Language** list. In this case, the existing correlation ID from the closest **cSOAP** connected to this component will be used. For more information about the **cSOAP** component, see cSOAP properties on page 53.
Correlation expression/Add Namespaces	This option appears when **XPath** is selected in the **Language** list. Select this check box to add namespaces for the Xpath expression. Click **[+]** to add as many

	namespaces as required to the table and define the prefix and URI in the corresponding columns.
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. **Note:** 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. **Note:** 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. **Note:** 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

Usage rule	**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.

cAMQP properties

cAMQP is used to exchange messages between a Route and a JMS provider using AMQP.

cAMQP sends messages to, or consumes messages from, a JMS Queue or Topic using the AMQP broker.

cAMQP Standard properties

These properties are used to configure cAMQP running in the Standard Job framework.

The Standard cAMQP component belongs to the Connectivity/Internet of Things and Connectivity/Messaging family.

Basic settings

URI/Type	Select the messaging type, either **queue** or **topic**.
URI/Destination	Type in a name for the message queue or topic in the message broker.
ConnectionFactory	Click **[...]** and select an MQ connection factory to be used for handling messages.

Advanced settings

Parameters	Set the optional parameters in the corresponding table. Click **[+]** as many times as required to add parameters to the table. Then click the corresponding **Value** field and enter a value. See the site http://camel.apache.org/amqp.html for available options. Be sure to set the `clientId` parameter to a unique value if you need to deploy multiple Routes using this component to Runtime. Otherwise the Routes will throw exception in Runtime.

Usage

Usage rule	**cAMQP** can be a start, middle or end component in a Route. It has to be used with the **cMQConnectionFactory** component, which creates a connection to a MQ server. For more information about **cMQConnectionFactory**, see cMQConnectionFactory properties on page 146.
Limitation	n/a

cAWSConnection properties

cAWSConnection is used to connect to Amazon Web Services for data storage and retrieval.

cAWSConnection establishes a connection to Amazon Web Services.

cAWSConnection Standard properties

These properties are used to configure cAWSConnection running in the Standard Job framework.

The Standard cAWSConnection component belongs to the AWS and Connectivity family.

Basic settings

Access Key	The Access Key ID that uniquely identifies an AWS Account. For how to get your Access Key and Access Secret, see Access keys (access key ID and secret access key).
Secret Key	The Secret Access Key, constituting the security credentials in combination with the access Key. To enter the secret key, click **[...]** next to the secret key field, and then in the pop-up dialog box enter the password between double quotes and click **OK** to save the settings.
Inherit credentials from AWS role	Select this check box to obtain AWS security credentials from Amazon EC2 instance metadata. To use this option, the Amazon EC2 instance must be started and your Route must be running on Amazon EC2. For more information, see Using an IAM Role to Grant Permissions to Applications Running on Amazon EC2 Instances , Instance Metadata and User Data, and IAM Roles for Amazon EC2.

Region	Specify the AWS region by selecting a region name from the list or entering a region between double quotation marks (e.g. "us-east-1") in the list. For more information about AWS Regions, see AWS Regions and Endpoints.

Advanced settings

Config Client	Select this check box to set the optional parameters for your AWS client in the corresponding table. Click **[+]** as many times as required and add the available parameters from the list to the table. Then click the corresponding **Value** field and enter a value. See the site Class ClientConfiguration for more information.

Usage

Usage rule	**cAWSConnection** cannot be added directly in a Route.

Related scenario:

For related scenarios, see:

- Scenario: Sending messages to and receiving messages from Amazon's S3 service on page 14
- Scenario: Sending Email using the cAWSSES component
- Scenario: Sending messages to Amazon's SNS topic on page 23
- Scenario: Sending messages to and receiving messages from Amazon's SQS queue on page 29

cAWSS3 properties

cAWSS3 is used for data storage and retrieval between a Route and the Amazon's S3 service.

cAWSS3 stores and retrieves objects from/to Amazon's Simple Storage Service (S3).

cAWSS3 Standard properties

These properties are used to configure cAWSS3 running in the Standard Job framework.

The Standard cAWSS3 component belongs to the AWS and Connectivity/File family.

Basic settings

Connection	Select an AWSS3 connection component from the list to reuse the connection details you already defined.
Bucket Name	Specify the name of the bucket, namely the top level folder, on Amazon's S3 service.

The following options are available only when the **cAWSS3** is used as a Producer:	
Storage Class	Select from **Standard**, **Standard - Infrequent Access** and **Reduced Redundancy**. For more information about storage classes, see the site http://docs.aws.amazon.com/AmazonS3/latest/dev/storage-class-intro.html.
Delete After Write	Select this check box to delete the local file object after it is uploaded to the S3 service.
Multi Part Upload	Select this check box to upload the file with multi part format, and specify part size in the corresponding field. The default size is 25M.
Server Side Encryption	Select this option to encrypt the object on the server side using the AWS-managed keys.
Message Headers	Set the message headers in the corresponding table. Click **[+]** as many times as required to add message headers to the table. Then click the corresponding **Value** field and enter a value. See the site http://camel.apache.org/aws-s3.html for available headers. Be sure to set the `CamelAwsS3Key` header which will be used as the name of the S3 file.
Use User Defined Headers	Select this check box to set user defined headers in the corresponding table. Click **[+]** as many times as required to add message headers to the table. Then click the corresponding **Value** field and enter a value. Note that user defined headers must start with `x-amz-meta-`.
The following options are available only when the **cAWSS3** is used as a Consumer:	

File Name	Specify the name of the file to be consumed.
Max Messages Per Poll	The maximum number of objects that can be retrieved in one poll.
Prefix	Specify the prefix of files so that only files with that prefix will be consumed.
Delete After Read	Select this check box to delete the object from the S3 service after it has been retrieved.
Include Body	Select this check box to include the exchange body in the content of the file. Otherwise only the headers will be set with the S3 object metadata and the body will be null.

Advanced settings

Advanced	Set the optional parameters in the corresponding table. Click **[+]** as many times as required to add parameters to the table. Then click the corresponding **Value** field and enter a value. See the site http://camel.apache.org/aws-s3.html for available options.

Usage

Usage rule	**cAWSS3** can be a start, middle or end component in a Route. It has to be used with the **cAWSConnection** component, which creates a connection to Amazon Web Services. For more information about **cAWSConnection**, see cAWSConnection properties on page 11.
Limitation	n/a

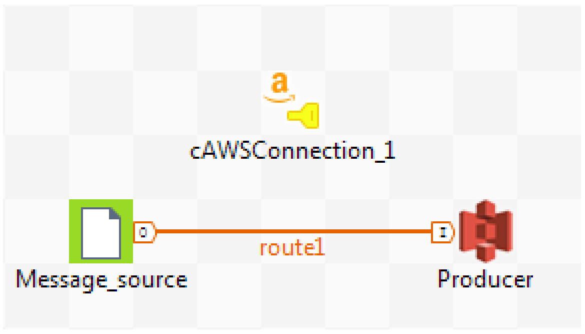
Scenario: Sending messages to and receiving messages from Amazon's S3 service

This scenario applies only to a Talend solution with ESB.

This scenario will show you how to use the **cAWSS3** component to send messages to and consume messages from Amazon's S3 service. To do this, two Routes are built, a message producer Route, and a consumer Route. Messages are sent to the Amazon's S3 service in the producer Route and then consumed in the consumer Route.

Building the producer Route

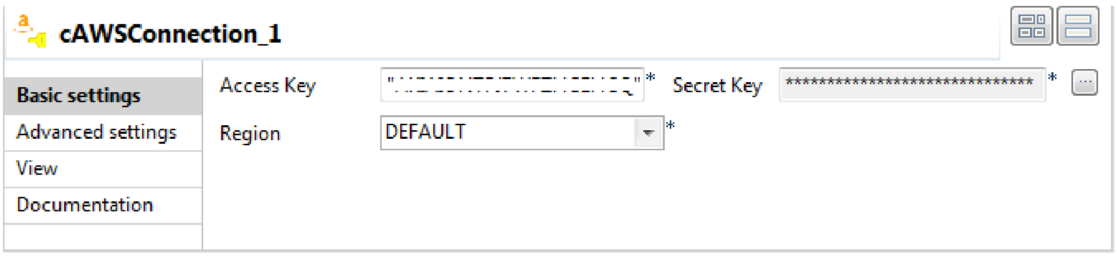
Dropping and linking the components



1. From the **Palette**, drag and drop a **cAWSConnection**, a **cFile**, and a **cAWSS3** component onto the design workspace.
2. Label the components for better identification of their roles and link them with the **Row** > **Route** connection as shown above.

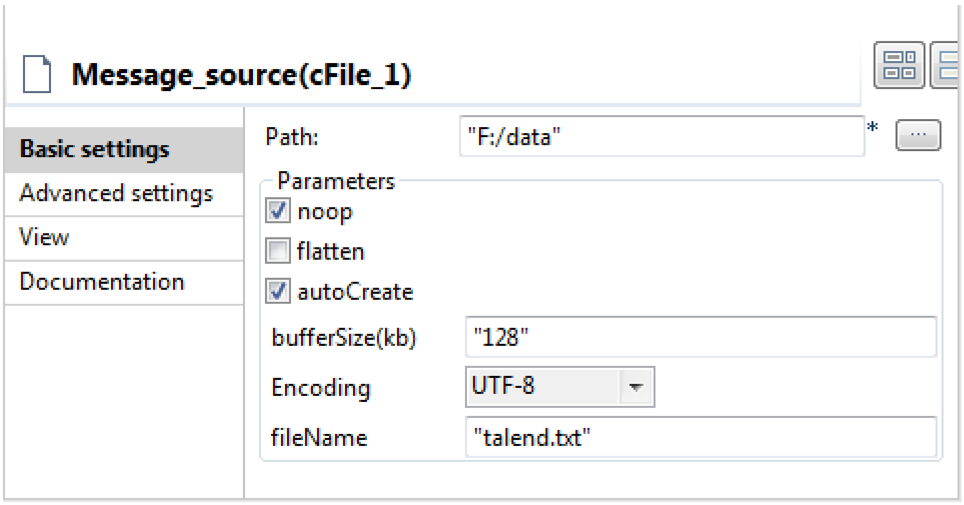
Configuring the components

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



Access Key "[redacted]"* Secret Key ******************************* *
Region DEFAULT *

2. In the **Access Key** and **Secret Key** fields, enter the authentication credentials of your AWS account. For how to get your Access Key and Access Secret, see Access keys (access key ID and secret access key).
3. Double-click the **cFile** component to open its **Basic settings** view in the **Component** tab.



Message_source(cFile_1)

Basic settings | Advanced settings | View | Documentation

Path: "F:/data"

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

bufferSize(kb): "128"

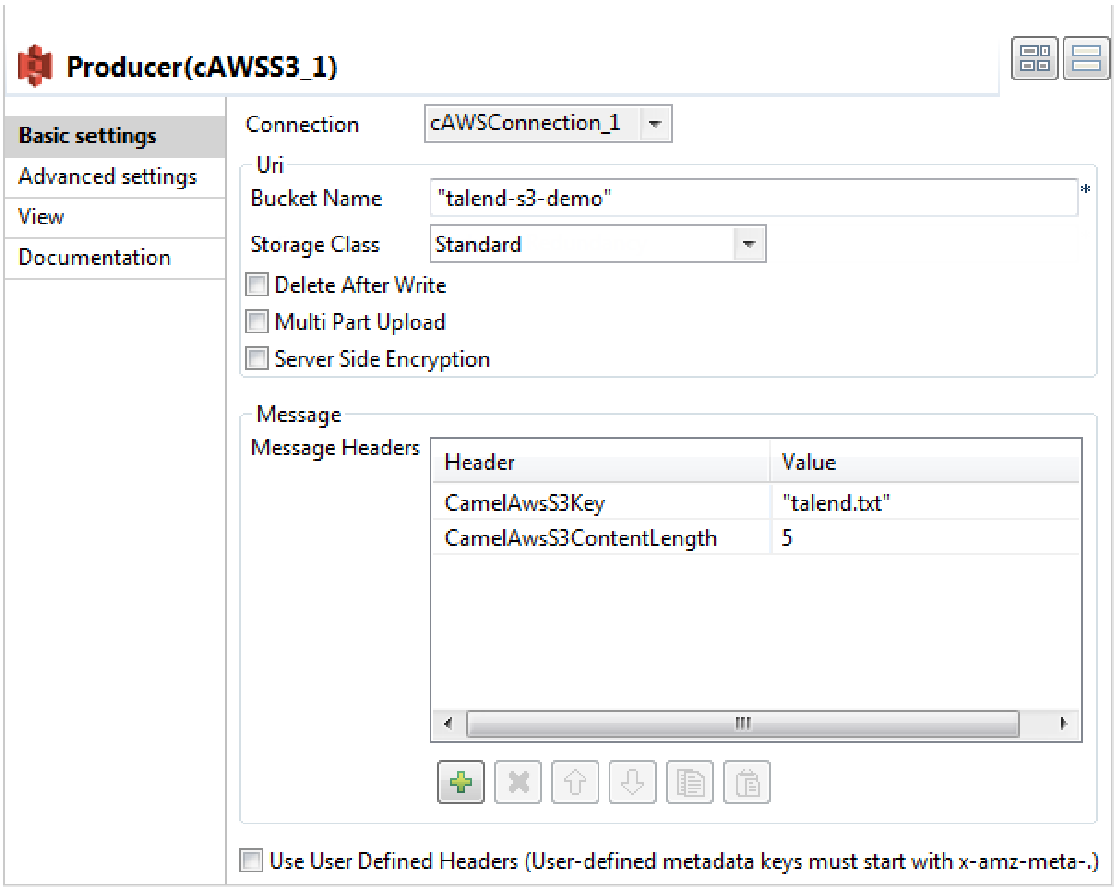
Encoding: UTF-8

fileName: "talend.txt"

4. In the **Path** field, browse to or enter the input file path.

   In the **fileName** field, enter the name of file to be uploaded to the S3 service.

   In this scenario, a TXT file with the name *talend* that contains a simple string *Hello World!* is used.
5. Double-click the **cAWSS3** component to open its **Basic settings** view in the **Component** tab.



Producer(cAWSS3_1)

Basic settings | Advanced settings | View | Documentation

Connection: cAWSConnection_1

Uri

Bucket Name: "talend-s3-demo"

Storage Class: Standard

- [ ] Delete After Write
- [ ] Multi Part Upload
- [ ] Server Side Encryption

Message

Message Headers

Header	Value
CamelAwsS3Key	"talend.txt"
CamelAwsS3ContentLength	5

- [ ] Use User Defined Headers (User-defined metadata keys must start with x-amz-meta-.)

6. In the **Connection** list, select the **cAWSConnection** component that you have just configured to connect to Amazon's S3 service.

   In the **Bucket Name** field, enter the name of the bucket to upload the file, `"talend-s3-demo"` in this use case.

   Under the **Message Headers** table, click **[+]** to add two rows in the table. Set the header **CamelAwsS3Key** with the value *"talend.txt"* as the name of the S3 object, and the header **CamelAwsS3ContentLength** with the value *5* to define the length of the object.

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

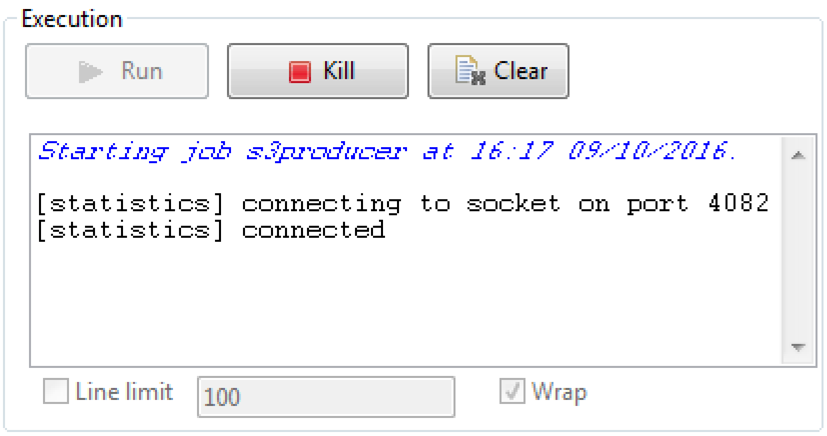
Viewing the code and executing the Route

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

```
from(
        "file://F:/data" + "?noop=true" + "&autoCreate=true"
                + "&flatten=false" + "&fileName=talend.txt"
                + "&charset=UTF-8" + "&bufferSize=128")
        .routeId("s3producer_cFile_1")
        .setHeader("CamelAwsS3Key", constant("talend.txt"))
        .setHeader("CamelAwsS3ContentLength", constant(5))
        .to("aws-s3:talend-s3-demo"
                + "?amazonS3Client=#conn_cAWSConnection_1")
        .id("s3producer_cAWSS3_1");
```

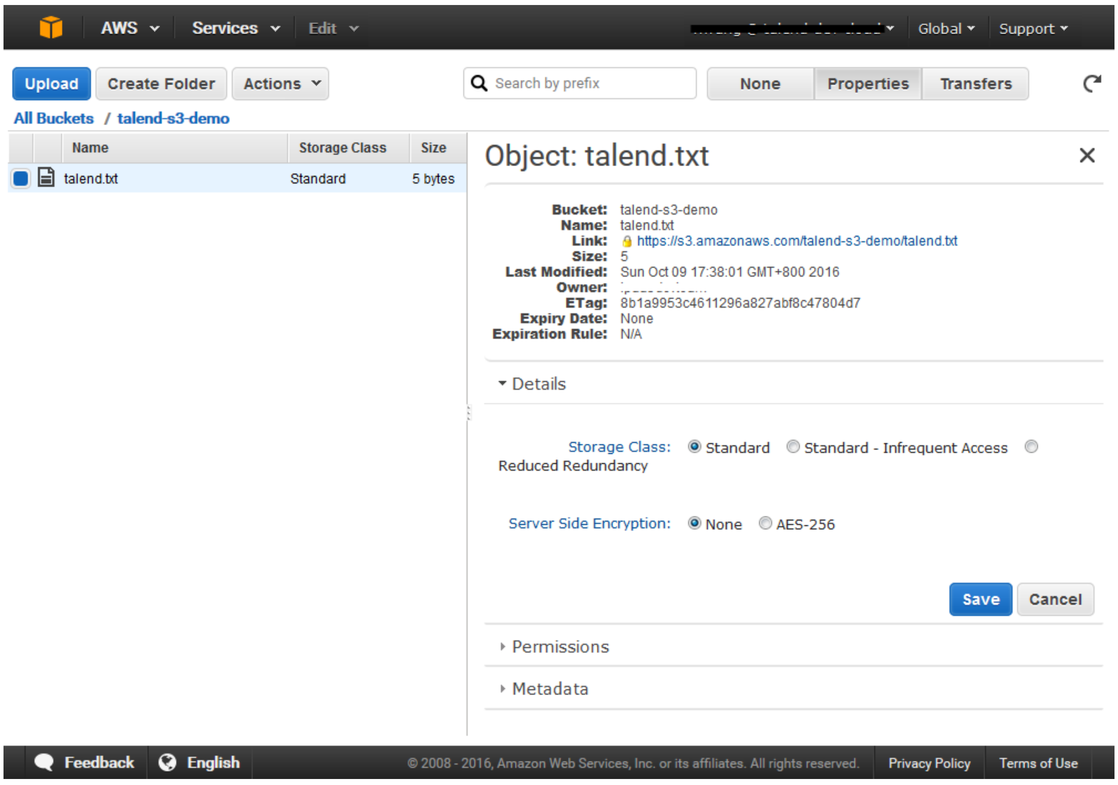
As shown above, the message is routed from `s3producer_cFile_1` to `s3producer_cAWSS3_1`.

2. Press **F6** to execute the Route. The logs of the message exchange are printed in the console.



```
Starting job s3producer at 16:17 09/10/2016.

[statistics] connecting to socket on port 4082
[statistics] connected
```

3. In the Amazon's S3 Web Console, you can see that the object *talend.txt* has been created.



Name	Storage Class	Size
talend.txt	Standard	5 bytes

Object: talend.txt

Bucket:	talend-s3-demo
Name:	talend.txt
Link:	https://s3.amazonaws.com/talend-s3-demo/talend.txt
Size:	5
Last Modified:	Sun Oct 09 17:38:01 GMT+800 2016
Owner:	[redacted]
ETag:	8b1a9953c4611296a827abf8c47804d7
Expiry Date:	None
Expiration Rule:	N/A

Details

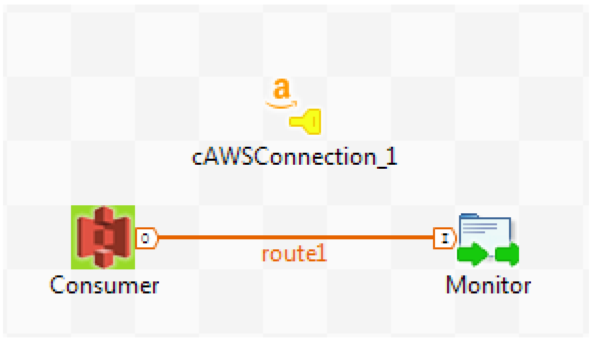
Storage Class: [● Standard] ○ Standard - Infrequent Access ○ Reduced Redundancy

Server Side Encryption: [● None] ○ AES-256

Save
Cancel
Permissions
Metadata
Feedback
English
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Building the consumer Route

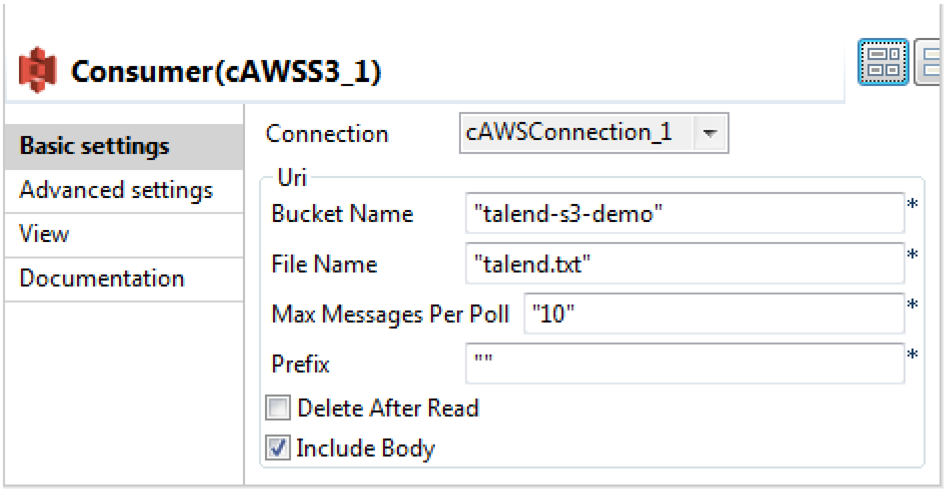
Arranging the flow of the message



1. From the **Palette**, drag and drop a **cAWSConnection**, a **cAWSS3** and a **cProcessor** component onto the design workspace.
2. Label the components for better identification of their roles and link them with the **Row** > **Route** connection as shown above.

Configuring how the message is processed

1. Configure the **cAWSConnection** using the same properties as in the producer Route.
2. Double-click the **cAWSS3** component to display its **Basic settings** view in the **Component** tab.



Connection: cAWSConnection_1

Uri

Bucket Name	"talend-s3-demo" *
File Name	"talend.txt" *
Max Messages Per Poll	"10" *
Prefix	"" *

☐ Delete After Read

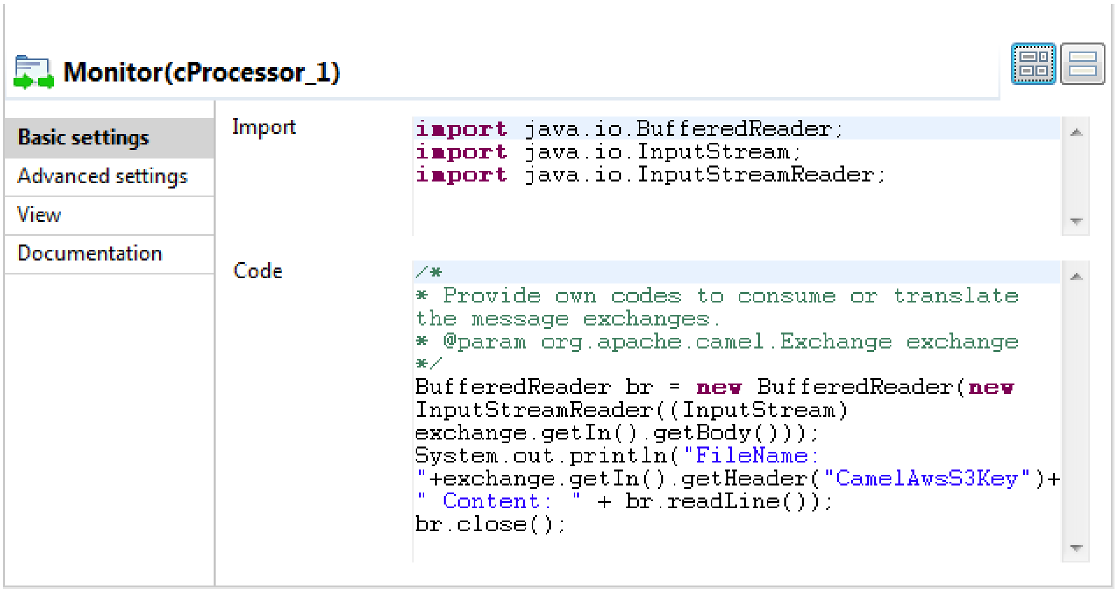
☑ Include Body

3. In the **Connection** list, select the **cAWSConnection** component to connect to Amazon's S3 service.

   In the **Bucket Name** field, enter the name of the bucket that contains the file to be consumed, `"talend-s3-demo"` in this use case.

   In the **File Name** field, enter the name of the file, *"talend.txt"*.

   Clear the **Delete After Read** check box to keep the S3 object file after it is consumed.
4. Double-click the **cProcessor** component to display its **Basic settings** view in the **Component** tab.



Import

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
```

Code

```
/*
* Provide own codes to consume or translate
the message exchanges.
* @param org.apache.camel.Exchange exchange
*/
BufferedReader br = new BufferedReader(new
InputStreamReader((InputStream)
exchange.getIn().getBody()));
System.out.println("FileName:
"+exchange.getIn().getHeader("CamelAwsS3Key")+
" Content: " + br.readLine());
br.close();
```

5. In the **Code** box, enter the following code to print the file name and its content in the execution console:`BufferedReader br = new BufferedReader(new InputStreamReader((InputStream) exchange.getIn().getBody())); System.out.println("FileName: "+exchange.getIn().getHeader("CamelAwsS3Key")+" Content: " + br.readLine()); br.close();`
6. Press **Ctrl+S** to save your Route.

Executing the Route

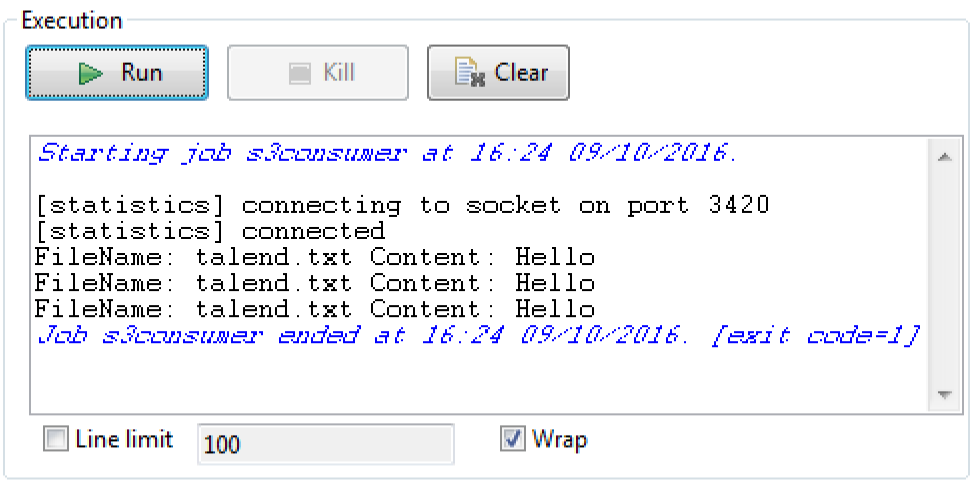
1. Click the **Code** tab at the bottom of the design workspace to check the generated code.

```
from(
        "aws-s3:talend-s3-demo"
                + "?amazonS3Client=#conn_cAWSConnection_1"
                + "&fileName=talend.txt" + "&deleteAfterRead=false")
        .routeId("s3consumer_cAWSS3_1")
        .process(new org.apache.camel.Processor() {
            public void process(org.apache.camel.Exchange exchange)
                    throws Exception {
                /*
                 * Provide own codes to consume or translate the message
                 * exchanges.
                 *
                 * @param org.apache.camel.Exchange exchange
                 */
                BufferedReader br = new BufferedReader(
                        new InputStreamReader((InputStream) exchange
                                .getIn().getBody()));
                System.out.println("FileName: "
                        + exchange.getIn().getHeader("CamelAwsS3Key")
                        + " Content: " + br.readLine());
                br.close();

            }

        }).id("s3consumer_cProcessor_1");
```

As shown above, the message flow is routed from `s3consumer_cAWSS3_1` and processed by `s3consumer_cProcessor_1`.

2. Press **F6** to execute the Route. The logs of the message exchange are printed in the console.

   The content of the file is `Hello` as the length of the object is defined to `5` when uploading the file to the S3 service.



```
Starting job s3consumer at 16:24 09/10/2016.

[statistics] connecting to socket on port 3420
[statistics] connected
FileName: talend.txt Content: Hello
FileName: talend.txt Content: Hello
FileName: talend.txt Content: Hello
Job s3consumer ended at 16:24 09/10/2016. [exit code=1]
```

cAWSSES properties

cAWSSES is used for sending emails with Amazon's SES service.

cAWSSES sends emails with Amazon's Simple Email Service (SES).

cAWSSES Standard properties

These properties are used to configure cAWSSES running in the Standard Job framework.

The Standard cAWSSES component belongs to the AWS and Connectivity/Messaging family.

Basic settings

Connection	Select an AWS connection component from the list to reuse the connection details you already defined.
Subject	Specify the email subject. It can be overriden by the `CamelAwsSesSubject` header.
From	Specify the sender's email address. It can be overriden by the `CamelAwsSesFrom` header.
To	Specify one or more destination email addresses. It can be overriden by the `CamelAwsSesTo` header.
Return Path	Specify the email address to which bounce notifications are to be forwarded. It can be overriden by the `CamelAwsSesReturnPath` header.
Reply to Addresses	Specify the reply-to email address(es) for the message. It can be overriden by the `CamelAwsSesReplyToAddresses` header.
Message Headers	Set the message headers in the corresponding table. Click **[+]** as many times as required to add message headers to the table. Then click the corresponding **Value** field and enter a value. See the site http://camel.apache.org/aws-ses.html for available headers.

Advanced settings

Advanced	Set the optional parameters in the corresponding table. Click **[+]** as many times as required to add parameters to the table. Then click the corresponding **Value** field and enter a value. See

	the site http://camel.apache.org/aws-ses.html for available options.

Usage

Usage rule	**cAWSSES** can be a start, middle or end component in a Route. It has to be used with the **cAWSConnection** component, which creates a connection to Amazon Web Services. For more information about **cAWSConnection**, see cAWSConnection properties on page 11.
Limitation	n/a

cAWSSNS properties

cAWSSNS is used for sending messages to an Amazon's Simple Notification topic.

cAWSSNS sends messages to an Amazon's Simple Notification topic.

cAWSSNS Standard properties

These properties are used to configure cAWSSNS running in the Standard Job framework.

The Standard cAWSSNS component belongs to the AWS and Connectivity/Messaging family.

Basic settings

Connection	Select an AWS connection component from the list to reuse the connection details you already defined.
Topic	Specify the name of the topic on Amazon's Simple Notification Service (SNS) to send message to.
Subject	Specify a subject for the message. It can be overriden by the message header `CamelAwsSnsSubject`.

Usage

Usage rule	**cAWSSNS** can only be an end component in a Route. It has to be used with the **cAWSConnection** component, which creates a connection to Amazon Web services. For more information about **cAWSConnection**, see cAWSConnection properties on page 11.
Limitation	n/a

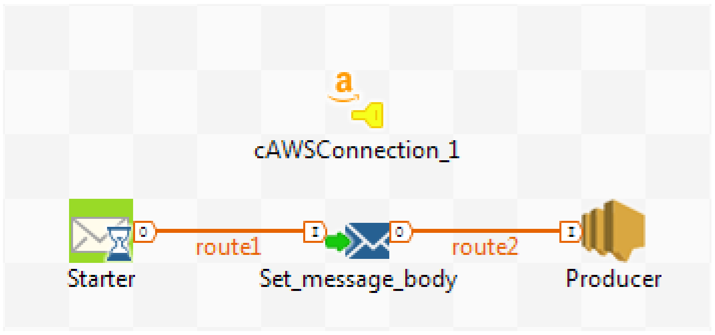
Scenario: Sending messages to Amazon's SNS topic

This scenario applies only to a Talend solution with ESB.

This scenario will show you how to use the **cAWSSNS** component to send message to the Amazon's SNS topic.

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon's SNS. For more information, see Amazon SNS.

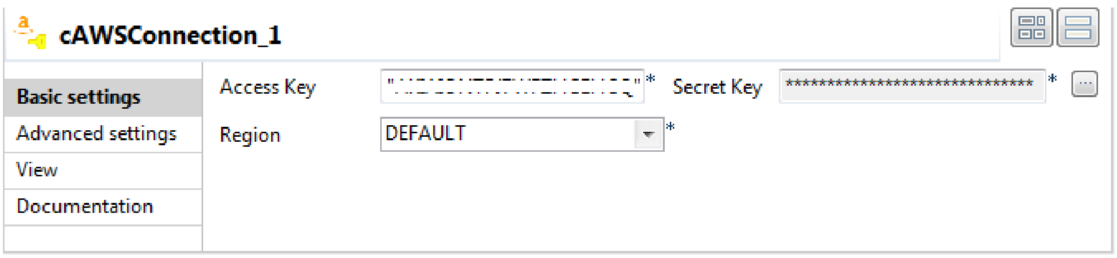
Dropping and linking the components



1. From the **Palette**, drag and drop a **cAWSConnection**, a **cTimer**, a **cSetBody**, and a **cAWSSNS** component onto the design workspace.
2. Label the components for better identification of their roles and link them with the **Row** > **Route** connection as shown above.

Configuring the components

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

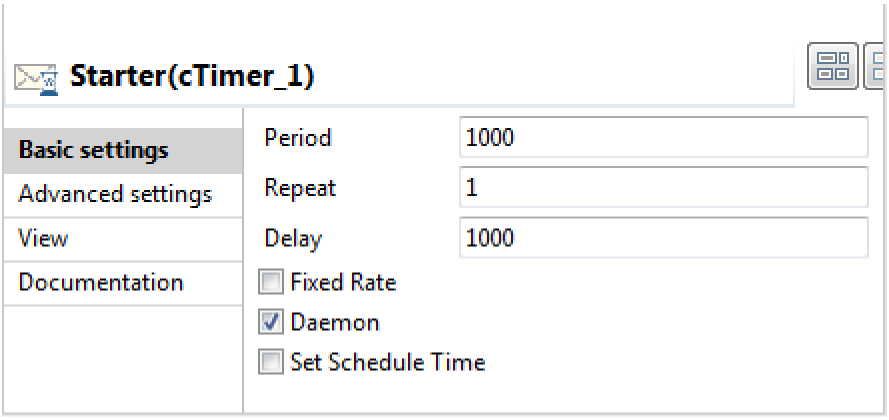


cAWSConnection_1

Basic settings
Advanced settings
View
Documentation

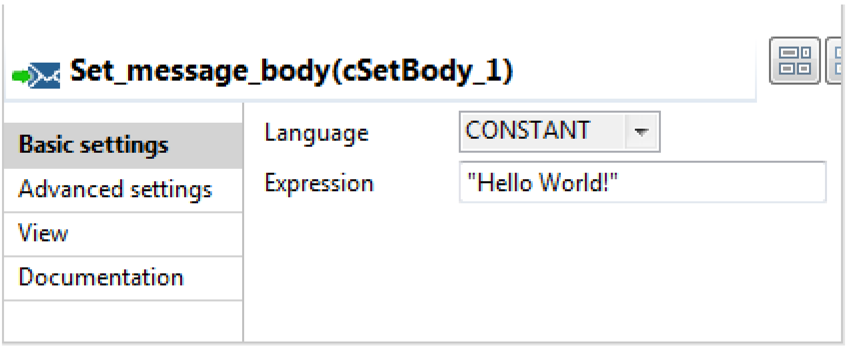
Access Key "[redacted]" *
Secret Key ****************************** *
Region DEFAULT *

2. In the **Access Key** and **Secret Key** fields, enter the authentication credentials of your AWS account. For how to get your Access Key and Access Secret, see Access keys (access key ID and secret access key).
3. Double-click the **cTimer** component to open its **Basic settings** view in the **Component** tab.


Starter(cTimer_1)

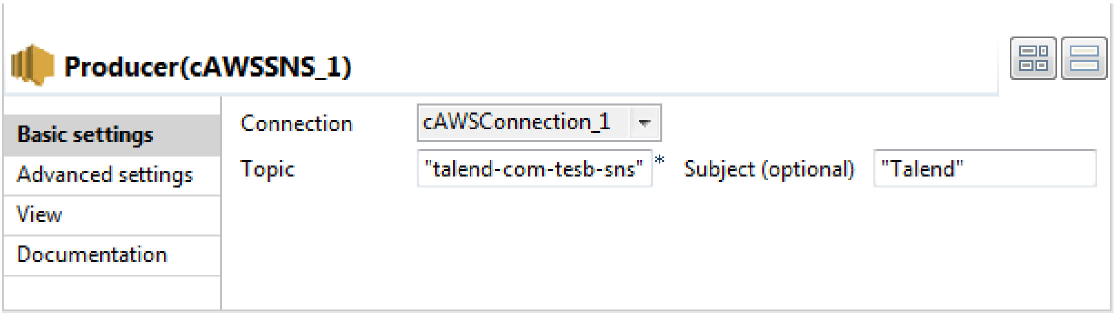
Period: 1000
Repeat: 1
Delay: 1000
Fixed Rate [ ]
Daemon [✓]
Set Schedule Time [ ]

4. In the **Repeat** field, enter `1` to generate the message exchange one time. Keep the default settings of the other options.
5. Double-click the **cSetBody** component to open its **Basic settings** view in the **Component** tab.


Set_message_body(cSetBody_1)

Language: CONSTANT
Expression: "Hello World!"

6. Select **CONSTANT** from the **Language** drop-down list and type in `"Hello world"` in the **Expression** field as the message body.
7. Double-click the **cAWSSNS** component to open its **Basic settings** view in the **Component** tab.


Producer(cAWSSNS_1)

Connection: cAWSConnection_1
Topic: "talend-com-tesb-sns" *
Subject (optional): "Talend"

8. In the **Connection** list, select the **cAWSConnection** component that you have just configured to connect to Amazon's SNS service.

   In the **Topic** field, enter the name of the topic to send message to, `"talend-com-tesb-sns"` in this use case.

   In the **Subject** field, give a subject for the message, for example `"Talend"`.
9. Press **Ctrl+S** to save your Route.

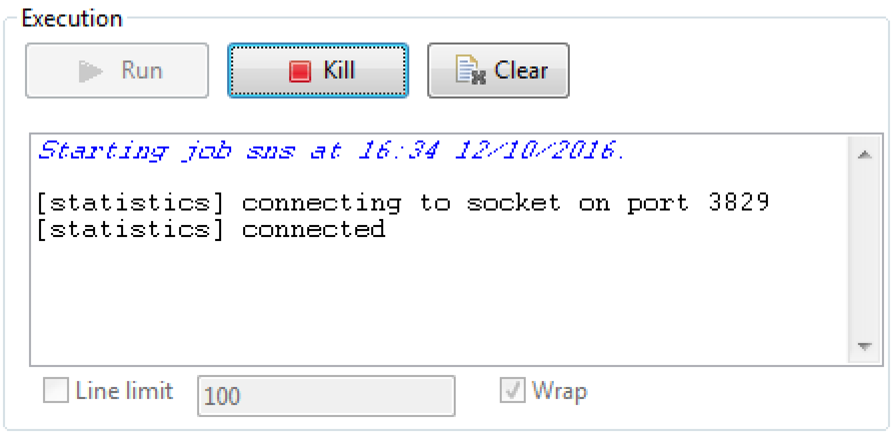
Viewing the code and executing the Route

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

```
from("timer:cTimer_1" + "?repeatCount=" + 1 + "&delay=" + 1000)
        .routeId("sns_cTimer_1")
        .setBody()
        .constant("Hello World!")
        .id("sns_cSetBody_1")
        .to("aws-sns:talend-com-tesb-sns"
                + "?amazonSNSClient=#conn_cAWSConnection_1"
                + "&subject=Talend").id("sns_cAWSSNS_1");
```

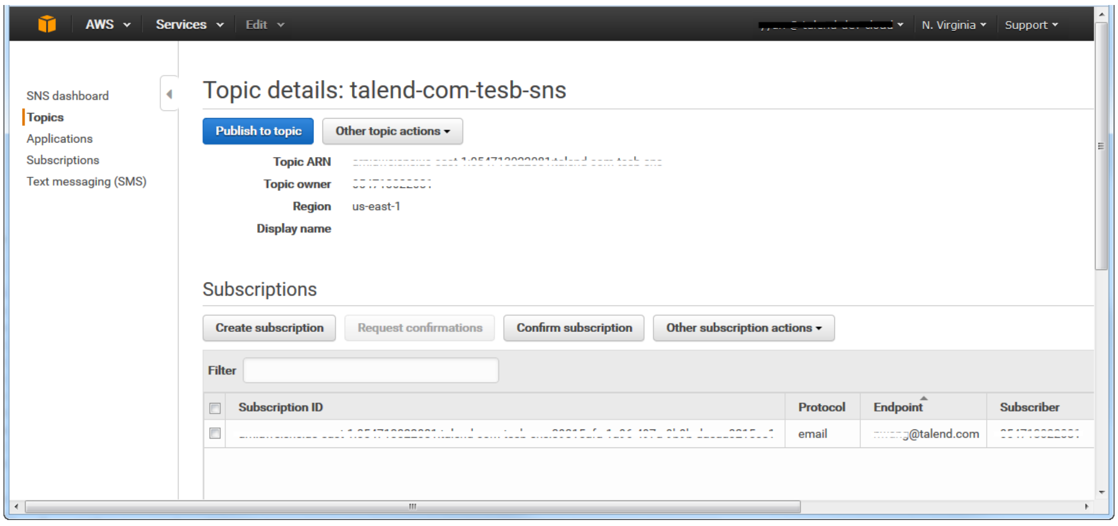
   As shown above, the message flow from `sns_cTimer_1` is given a payload by `sns_cSetBody_1` and then sent to `sns_cAWSSNS_1`.

2. Press **F6** to execute the Route. The logs of the message exchange are printed in the console.



```
Starting job sns at 16:34 12/10/2016.

[statistics] connecting to socket on port 3829
[statistics] connected
```

3. An email address has already subscribed to the SNS topic as shown below. For more information about how to subscribe to a topic, see the site http://docs.aws.amazon.com/sns/latest/dg/SubscribeTopic.html.



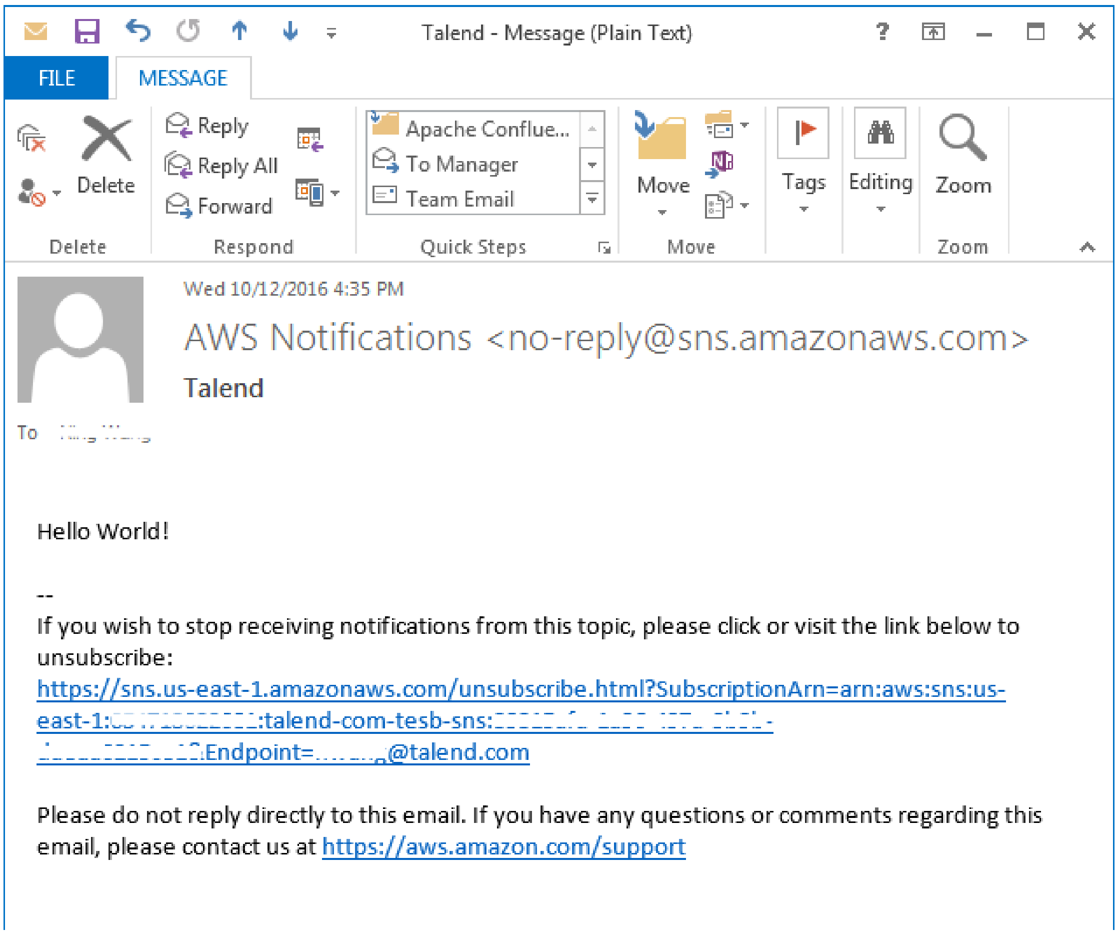
Topic details: talend-com-tesb-sns

Topic ARN	[redacted]
Topic owner	[redacted]
Region	us-east-1
Display name	

Subscriptions

Subscription ID	Protocol	Endpoint	Subscriber
[redacted]	email	[redacted]@talend.com	[redacted]

An notification of the message is sent to the Email address:



Wed 10/12/2016 4:35 PM

AWS Notifications <no-reply@sns.amazonaws.com>

Talend

To [redacted]

Hello World!

--
If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:
https://sns.us-east-1.amazonaws.com/unsubscribe.html?SubscriptionArn=arn:aws:sns:us-east-1:[redacted]:talend-com-tesb-sns:[redacted]&Endpoint=[redacted]@talend.com

Please do not reply directly to this email. If you have any questions or comments regarding this email, please contact us at https://aws.amazon.com/support

cAWSSQS properties

cAWSSQS provide connectivity to Amazon's SQS service. It is used for sending and receiving messages between a Route and the Amazon's SQS service.

cAWSSQS sends and receives messages to/from Amazon's Simple Queue Service (SQS).

cAWSSQS Standard properties

These properties are used to configure cAWSSQS running in the Standard Job framework.

The Standard cAWSSQS component belongs to the AWS and Connectivity/Messaging family.

Basic settings

Connection	Select an AWS connection component from the list to reuse the connection details you already defined.
Queue Name	Enter the name of the queue to send message to or receive message from. When the **cAWSSQS** is used as a producer, the queue will be created if it does not exist. Queue names must be

	1-80 characters in length and be composed of alphanumeric characters, hyphens (-), and underscores (_).
The following options are available only when the **cAWSSQS** is used as a Producer:	
Delay (in seconds)	Specify the amount of time to delay the first delivery of all messages added to this queue.
Wait Time (0 to 20 seconds)	Specify the maximum amount of time in seconds (0 to 20) that a long polling receive call will wait for a message to become available before returning an empty response.
The following options are available only when the **cAWSSQS** is used as a Consumer:	
Delete Message / After Read (processed by route)	Select this check box to delete the message from the queue after it is read and processed by the Route.
Delete Message / If Filtered (matched by filter)	Select this check box to delete the messages from the queue that are filtered in the Route, even if the exchange fails to get through a filter.
Allow Multiple Threads	Select this check box to allow multiple threads to poll the SQS queue to increase throughput. When this option is enabled, you need to specify the maximum number of concurrent consumers and objects that can be retrieved in one poll in the **Concurrency Number** and **Max Messages Per Poll** fields respectively.
Extend Message Visibility	Select this check box to enable a scheduled background task to keep extending the message visibility on SQS. This is needed if it takes a long time to process the message. When this option is enabled, you need to set the duration

	in seconds that the received messages are hidden from subsequent retrieve requests in the **Visibility Timeout** field. For more information, see the site http://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/API_ChangeMessageVisibility.html.
Request Attribute / All	Select this check box to retrieve the standard Amazon SQS attributes along with each message. For more information about the Amazon SQS attributes, see ReceiveMessage > Request Parameters.
Request Attribute / Approximate First Receive Timestamp	Select this check box to retrieve the `ApproximateFirstReceiveTimestamp` attribute only along with each message. For more information about the Amazon SQS attributes, see ReceiveMessage > Request Parameters.
Request Attribute / Approximate Receive Count	Select this check box to retrieve the `ApproximateReceiveCount` attribute only along with each message. For more information about the Amazon SQS attributes, see ReceiveMessage > Request Parameters.
Request Attribute / Sender ID	Select this check box to retrieve the `SenderId` attribute only along with each message. For more information about the Amazon SQS attributes, see ReceiveMessage > Request Parameters.
Request Attribute / Sent Timestamp	Select this check box to retrieve the `SentTimestamp` attribute only along with each message. For more information about the Amazon SQS attributes, see ReceiveMessage > Request Parameters.

Advanced settings

AWS Account ID (Queue Owner)	Specify the queue owner's AWS account ID when you need to connect the queue with different account owner.
Queue Configuration / Attributes	Set the optional queue attributes in the corresponding table. Click **[+]** as many times as required to add attributes to the table. Then click the corresponding **Value** field and enter a value. See the site http://camel.apache.org/aws-sqs.html for available options.
Request Message Attribute	This option is only available when **cAWSSQS** is used as a Consumer. Select this check box and add the attribute to be retrieved along with each message.

Usage

Usage rule	**cAWSSQS** can be a start, middle or end component in a Route. It has to be used with the **cAWSConnection** component, which creates a connection to Amazon SNS service. For more information about **cAWSConnection**, see cAWSConnection properties on page 11.
Limitation	n/a

Scenario: Sending messages to and receiving messages from Amazon's SQS queue

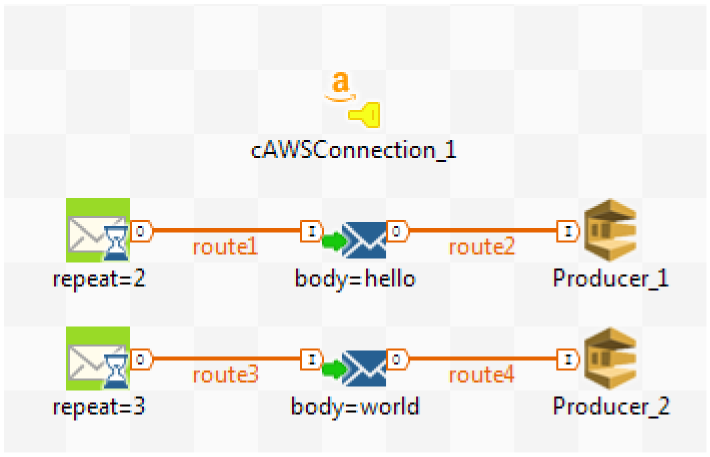
This scenario applies only to a Talend solution with ESB.

This scenario will show you how to use the **cAWSSQS** component to send messages to and consume messages from an SQS queue. To do this, two Routes are built, a message producer Route, and a consumer Route. Messages are sent to the SQS queue in the producer Route and then consumed in the consumer Route.

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon SQS. For more information, see Amazon SQS.

Building the producer Route

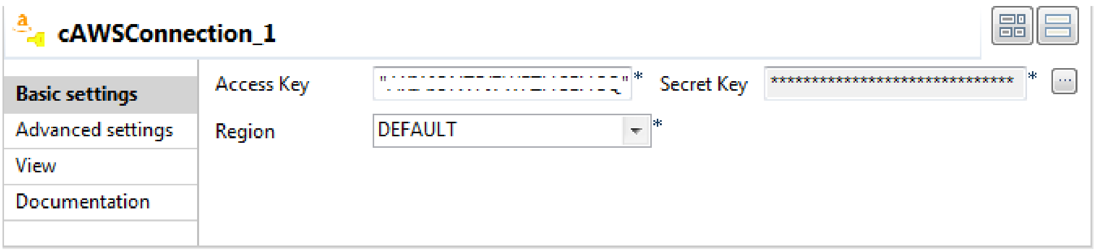
Dropping and linking the components



1. From the **Palette**, drag and drop a **cAWSConnection**, two **cTimer**, two **cSetBody**, two **cAWSSQS** components onto the design workspace.
2. Label the components for better identification of their roles and link them with the **Row** > **Route** connection as shown above.

Configuring the components

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

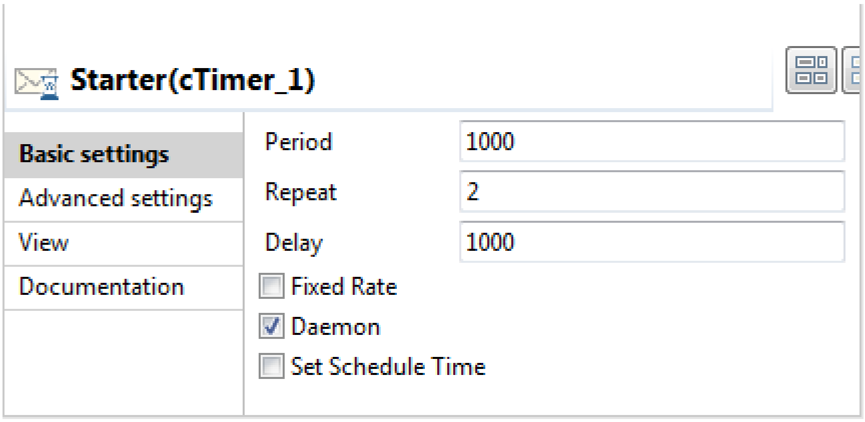


cAWSConnection_1

Basic settings | Advanced settings | View | Documentation

Access Key: [redacted] *
Secret Key: "******************************" *
Region: DEFAULT *

2. In the **Access Key** and **Secret Key** fields, enter the authentication credentials of your AWS account. For how to get your Access Key and Access Secret, see Access keys (access key ID and secret access key).
3. Double-click the **cTimer** labelled *repeat=2* to open its **Basic settings** view in the **Component** tab.

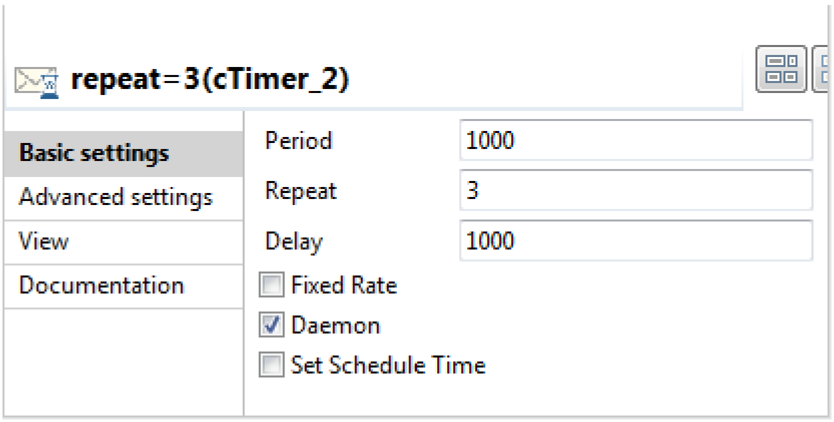


Starter(cTimer_1)

Basic settings | Advanced settings | View | Documentation

Period: 1000
Repeat: 2
Delay: 1000
☐ Fixed Rate
☑ Daemon
☐ Set Schedule Time

4. In the **Repeat** field, enter 2 to generate the message exchange twice. Keep the default settings of the other options.
5. Configure the **cTimer** labelled *repeat=3* in the same way to generate the message exchange three times.

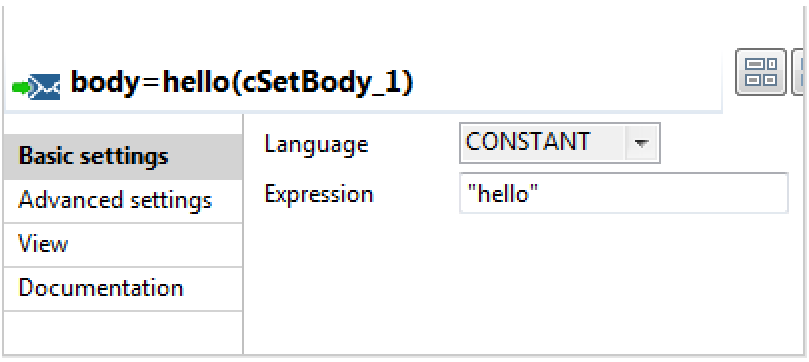


repeat=3(cTimer_2)

Period	1000
Repeat	3
Delay	1000

☐ Fixed Rate
☑ Daemon
☐ Set Schedule Time

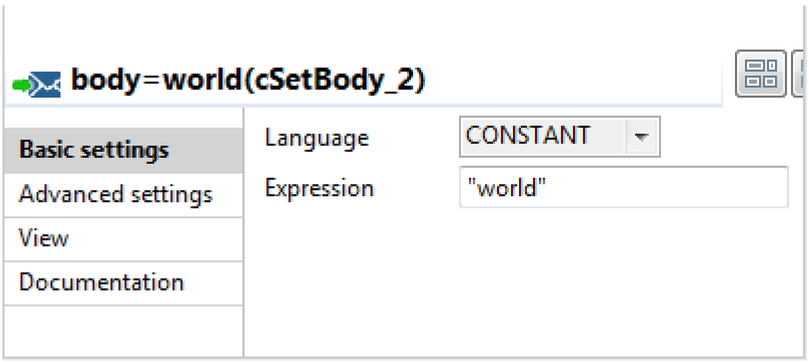
6. Double-click the **cSetBody** component labelled *body=hello* to open its **Basic settings** view in the **Component** tab.
7. Select **CONSTANT** from the **Language** list and type in *hello* in the **Expression** field as the message body.



body=hello(cSetBody_1)

Language	CONSTANT
Expression	"hello"

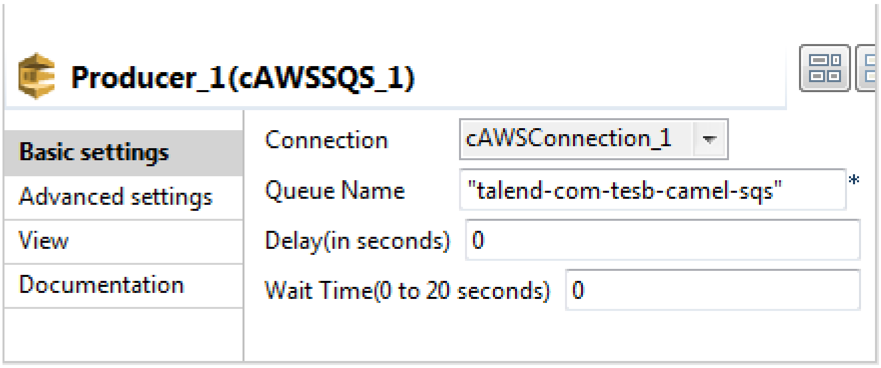
8. Configure the **cSetBody** labelled *body=world* in the same way to set the message body as *world*.



body=world(cSetBody_2)

Language	CONSTANT
Expression	"world"

9. Double-click the **cAWSSQS** labelled *Producer_1* to open its **Basic settings** view in the **Component** tab.



Producer_1(cAWSSQS_1)

Basic settings
Advanced settings
View
Documentation

Connection: cAWSConnection_1
Queue Name: "talend-com-tesb-camel-sqs" *
Delay(in seconds): 0
Wait Time(0 to 20 seconds): 0

10. In the **Connection** list, select the **cAWSConnection** component that you have just configured to connect to Amazon's SQS service.

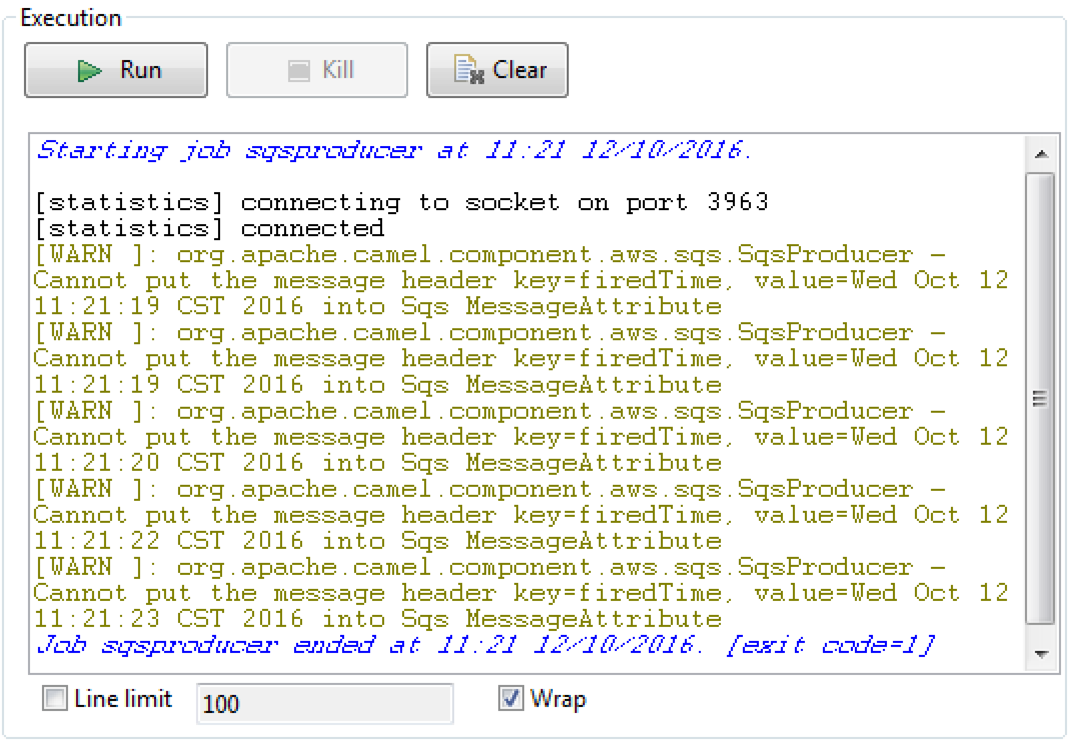
    In the **Queue Name** field, type in the name of the SQS queue to send the messages.
11. Configure the **cAWSSQS** labelled *Producer_2* with the same properties to send the messages to the same queue.
12. Press **Ctrl+S** to save your Route.

Viewing the code and executing the Route

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

```
from("timer:cTimer_1" + "?repeatCount=" + 2 + "&delay=" + 1000)
        .routeId("sqsproducer_cTimer_1")
        .setBody()
        .constant("hello")
        .id("sqsproducer_cSetBody_1")
        .to("aws-sqs:talend-com-tesb-camel-sqs"
                + "?amazonSQSClient=#cAWSSQS_cAWSConnection_1")
        .id("sqsproducer_cAWSSQS_1");
from("timer:cTimer_2" + "?repeatCount=" + 3 + "&delay=" + 1000)
        .routeId("sqsproducer_cTimer_2")
        .setBody()
        .constant("world")
        .id("sqsproducer_cSetBody_2")
        .to("aws-sqs:talend-com-tesb-camel-sqs"
                + "?amazonSQSClient=#cAWSSQS_cAWSConnection_1")
        .id("sqsproducer_cAWSSQS_2");
```

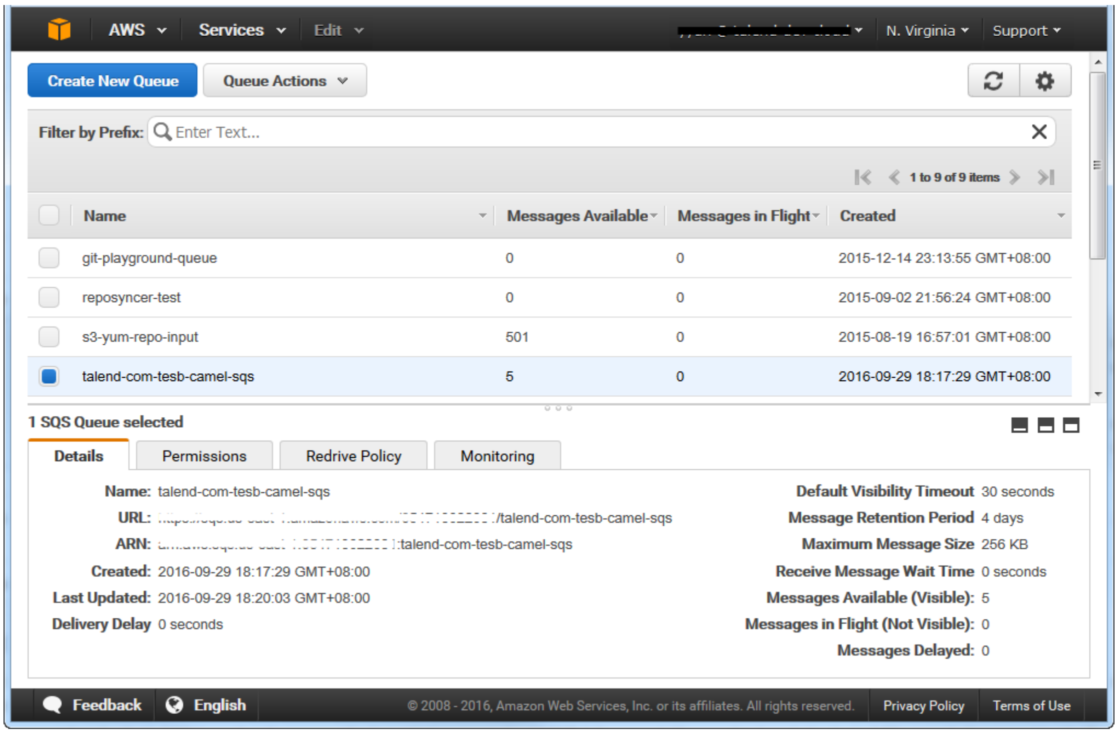
   As shown above, in the first sub-route, the message flow from `sqsproducer_cTimer_1` is given a payload by `sqsproducer_cSetBody_1` and then sent to `sqsproducer_cAWSSQS_1`. The second sub-route is in the same way.
2. Press **F6** to execute the Route. The logs of the message exchange are printed in the console. The warn message shows that header `fireTime` is not put into the message attribute.



```
Starting job sqsproducer at 11:21 12/10/2016.

[statistics] connecting to socket on port 3963
[statistics] connected
[WARN ]: org.apache.camel.component.aws.sqs.SqsProducer -
Cannot put the message header key=firedTime, value=Wed Oct 12
11:21:19 CST 2016 into Sqs MessageAttribute
[WARN ]: org.apache.camel.component.aws.sqs.SqsProducer -
Cannot put the message header key=firedTime, value=Wed Oct 12
11:21:19 CST 2016 into Sqs MessageAttribute
[WARN ]: org.apache.camel.component.aws.sqs.SqsProducer -
Cannot put the message header key=firedTime, value=Wed Oct 12
11:21:20 CST 2016 into Sqs MessageAttribute
[WARN ]: org.apache.camel.component.aws.sqs.SqsProducer -
Cannot put the message header key=firedTime, value=Wed Oct 12
11:21:22 CST 2016 into Sqs MessageAttribute
[WARN ]: org.apache.camel.component.aws.sqs.SqsProducer -
Cannot put the message header key=firedTime, value=Wed Oct 12
11:21:23 CST 2016 into Sqs MessageAttribute
Job sqsproducer ended at 11:21 12/10/2016. [exit code=1]
```

Line limit 100 [✓ Wrap]

3. In the SQS Web Console, you can see that there are now 5 messages in the queue *talend-com-tesb-camel-sqs*.



	Name	Messages Available	Messages in Flight	Created
	git-playground-queue	0	0	2015-12-14 23:13:55 GMT+08:00
	reposyncer-test	0	0	2015-09-02 21:56:24 GMT+08:00
	s3-yum-repo-input	501	0	2015-08-19 16:57:01 GMT+08:00
✓	talend-com-tesb-camel-sqs	5	0	2016-09-29 18:17:29 GMT+08:00

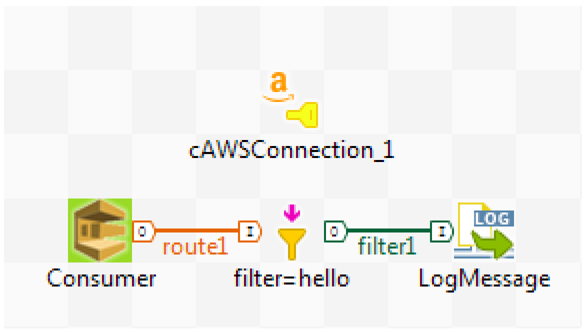
1 SQS Queue selected

Name:	talend-com-tesb-camel-sqs	Default Visibility Timeout	30 seconds
URL:	[redacted]/talend-com-tesb-camel-sqs	Message Retention Period	4 days
ARN:	[redacted]:talend-com-tesb-camel-sqs	Maximum Message Size	256 KB
Created:	2016-09-29 18:17:29 GMT+08:00	Receive Message Wait Time	0 seconds
Last Updated:	2016-09-29 18:20:03 GMT+08:00	Messages Available (Visible):	5
Delivery Delay	0 seconds	Messages in Flight (Not Visible):	0
		Messages Delayed:	0

Feedback
English
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Building the consumer Route

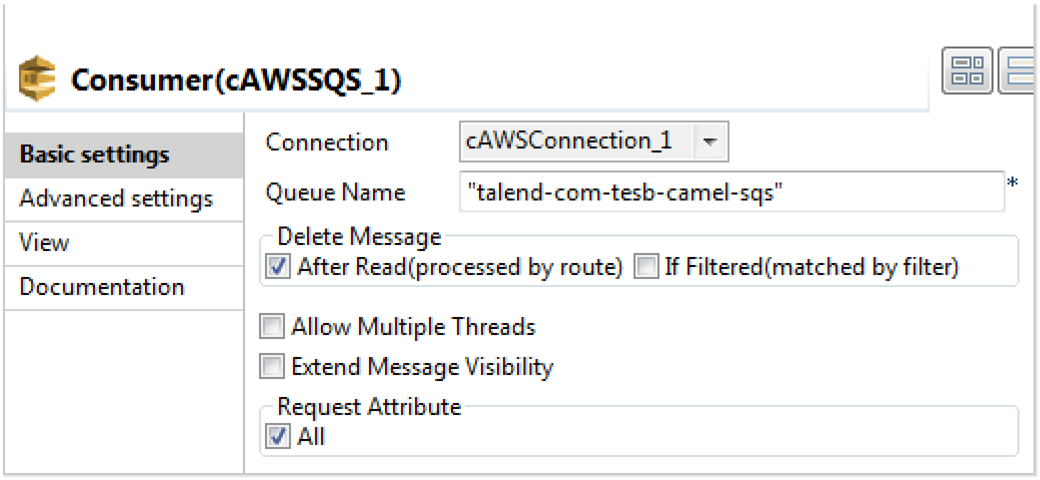
Arranging the flow of the message



1. From the **Palette**, drag and drop a **cAWSConnection**, a **cAWSSQS**, a **cMessageFilter** and a **cLog** component onto the design workspace.
2. Link the **cAWSSQS** to **cMessageFilter** with the **Row** > **Route** connection and **cMessageFilter** to **cLog** with the **Row** > **Filter** connection.
3. Label the components for better identification of their roles.

Configuring how the message is processed

1. Configure the **cAWSConnection** using the same properties as in the producer Route.
2. Double-click the **cAWSSQS** component to display its **Basic settings** view in the **Component** tab.



Consumer(cAWSSQS_1)

Connection: cAWSConnection_1

Queue Name: "talend-com-tesb-camel-sqs" *

Delete Message: [✓] After Read(processed by route) [ ] If Filtered(matched by filter)

[ ] Allow Multiple Threads

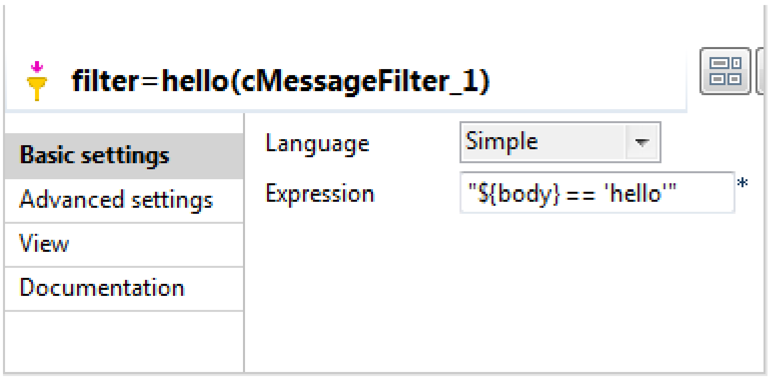
[ ] Extend Message Visibility

Request Attribute: [✓] All

3. In the **Connection** list, select the **cAWSConnection** component to connect to Amazon's SQS service.

   In the **Queue Name** field, enter the name of the queue to consume the message from.

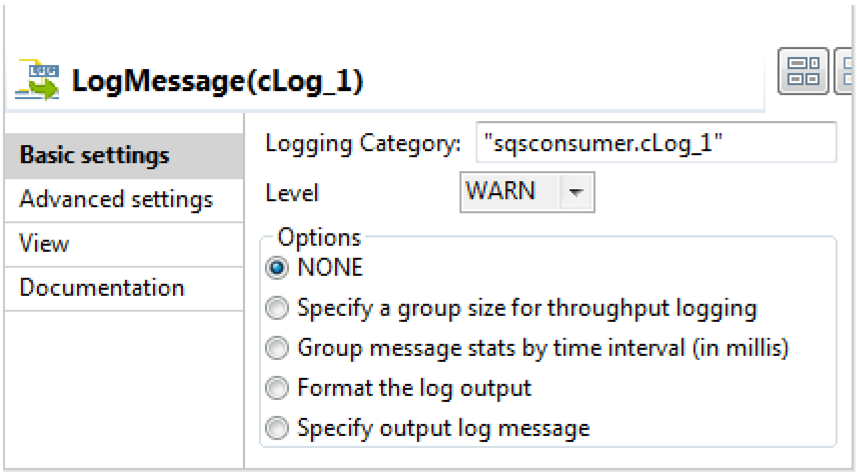
   Select the **Delete Message / After Read** check box and clear the **If Filtered** check box to delete the messages that are read and keep those that are not consumed.
4. Double-click the **cMessageFilter** component to display its **Basic settings** view in the **Component** tab.



Language: Simple

Expression: "${body} == 'hello'" *

5. Select **Simple** from the **Language** list and enter *"${body} == 'hello'"* in the **Expression** field to sort out messages with *hello* as the message body.
6. Keep the default settings of the **cLog** component to log the message exchanges.



Logging Category: "sqsconsumer.cLog_1"

Level: WARN

Options: [● NONE] Specify a group size for throughput logging / Group message stats by time interval (in millis) / Format the log output / Specify output log message

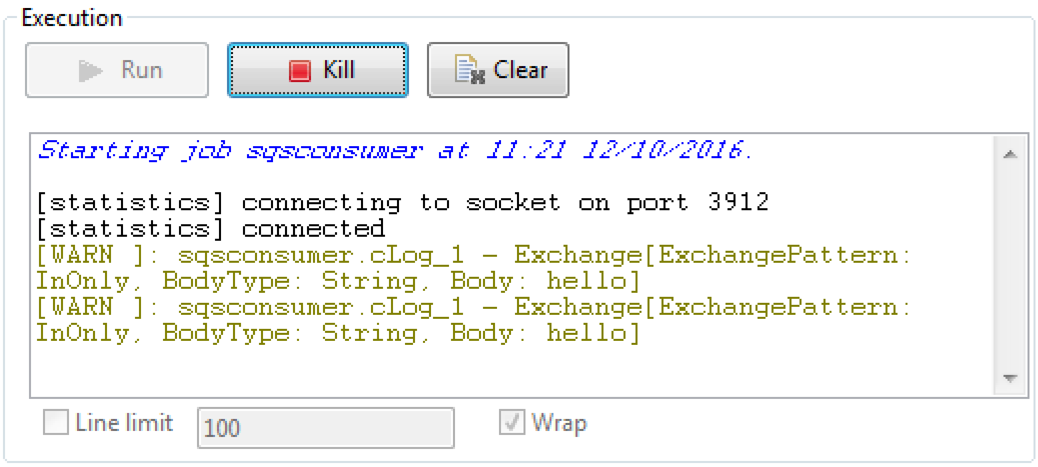
7. Press **Ctrl+S** to save your Route.

Executing the Route

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

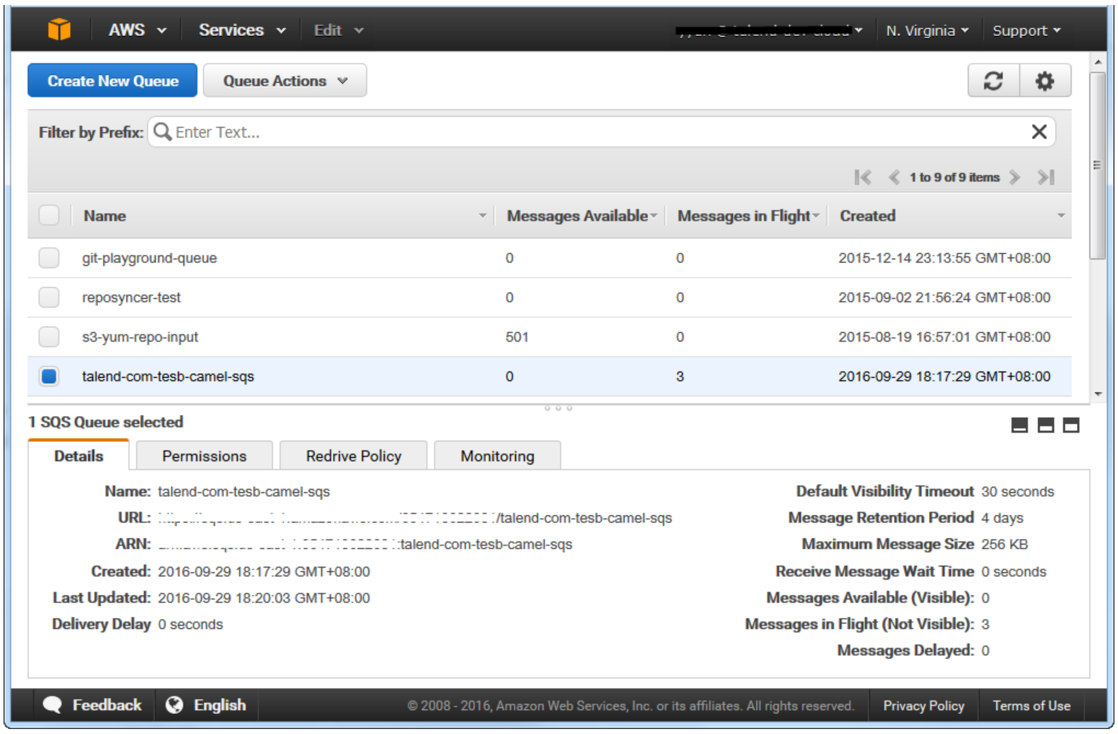
```
from(
        "aws-sqs:talend-com-tesb-camel-sqs"
                + "?amazonSQSClient=#cAWSSQS_cAWSConnection_1"
                + "&deleteIfFiltered=" + false + "&attributeNames=All")
        .routeId("sqsconsumer_cAWSSQS_1").filter()
        .simple("${body} == 'hello'")
        .id("sqsconsumer_cMessageFilter_1")
        .to("log:sqsconsumer.cLog_1" + "?level=WARN")

        .id("sqsconsumer_cLog_1");
```

   As shown above, the message flow from `sqsconsumer_cAWSSQS_3`, is filtered by `sqsconsumer_cMessageFilter_1`, and then routed to `sqsconsumer_cLog_1`.
2. Press **F6** to execute the Route. The messages with the body `hello` are consumed and are printed in the console.



```
Starting job sqsconsumer at 11:21 12/10/2016.

[statistics] connecting to socket on port 3912
[statistics] connected
[WARN ]: sqsconsumer.cLog_1 - Exchange[ExchangePattern:
InOnly, BodyType: String, Body: hello]
[WARN ]: sqsconsumer.cLog_1 - Exchange[ExchangePattern:
InOnly, BodyType: String, Body: hello]
```

3. In the SQS Web Console, you can see that there are now 3 three messages left in the queue *talend-com-tesb-camel-sqs* . The two messages with the body `hello` have been consumed and deleted as configured in the **cAWSSQS** component.



	Name	Messages Available	Messages in Flight	Created
	git-playground-queue	0	0	2015-12-14 23:13:55 GMT+08:00
	reposyncer-test	0	0	2015-09-02 21:56:24 GMT+08:00
	s3-yum-repo-input	501	0	2015-08-19 16:57:01 GMT+08:00
✓	talend-com-tesb-camel-sqs	0	3	2016-09-29 18:17:29 GMT+08:00

1 SQS Queue selected

Details | Permissions | Redrive Policy | Monitoring

Name:	talend-com-tesb-camel-sqs
URL:	[redacted]/talend-com-tesb-camel-sqs
ARN:	[redacted]:talend-com-tesb-camel-sqs
Created:	2016-09-29 18:17:29 GMT+08:00
Last Updated:	2016-09-29 18:20:03 GMT+08:00
Delivery Delay	0 seconds
Default Visibility Timeout	30 seconds
Message Retention Period	4 days
Maximum Message Size	256 KB
Receive Message Wait Time	0 seconds
Messages Available (Visible):	0
Messages in Flight (Not Visible):	3
Messages Delayed:	0

cBean properties

cBean allows you to call a predefined Java bean.

cBean invokes a Java bean that is stored in the **Code** node of the **Repository** or registered by a **cBeanRegister**.

cBean Standard properties

These properties are used to configure cBean running in the Standard Job framework.

The Standard cBean component belongs to the Custom family.

Basic settings

Reference	Select this option to reference a Java bean registered by a **cBeanRegister**. In the **Id** field that appears, enter the Id of the Java bean.
New Instance	Select this option to invoke a Java bean that is stored in the **Code** node of the **Repository**. In the **Bean class** field that appears, enter the name of the bean class. For more information about creating and using Java Beans, see *Talend Studio User Guide*.
Specify the method	Select this check box to enter the name of a method to be included in the bean.

Usage

Usage rule	**cBean** can be a start, middle or end component in a Route.
Limitation	n/a

Related Scenario

For a related scenario, see:

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

cBeanRegister properties

cBeanRegister allows you to register a Java bean in the registry to be used in message exchanges.

cBeanRegister registers a Java bean in the registry that can be called on using the ID of the bean in message exchanges.

cBeanRegister Standard properties

These properties are used to configure cBeanRegister running in the Standard Job framework.

The Standard cBeanRegister component belongs to the Custom family.

Basic settings

Id	Enter any string which is used to look up the bean in the registry.
Simple	Select this option to call a bean class that is stored in the **Code** node of the **Repository**.
Customized	Select this option to define the Java bean by entering the code in the **Code** box.
Class Name	This field appears when the **Simple** option is selected. Enter the name of the bean class that is stored in the **Code** node of the **Repository**. For more information about creating and using Java Beans, see *Talend Studio User Guide*.
Specify Arguments	This check box appears when the **Simple** option is selected. Select this check box to set the optional arguments in the corresponding table. Click **[+]** as many times as required to add arguments to the table.
Imports	This box appears when the **Customized** option is selected. Enter the Java code that helps to import, if necessary, external libraries used in the **Code** box.
Code	This box appears when the **Customized** option is selected. Enter the code of the bean in the box.

Usage

Usage rule	**cBeanRegister** cannot be added directly in a Route.
Limitation	n/a

Related Scenario

For a related scenario, see:

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

cConfig properties

cConfig allows you to set the CamelContext using Java code.

cConfig manipulates the CamelContext, which is the single routing rule base in a Route.

cConfig Standard properties

These properties are used to configure cConfig running in the Standard Job framework.

The Standard cConfig component belongs to the Custom family.

Basic settings

Imports	Enter the Java code that helps to import, if necessary, external libraries used in the **Code** box.
Code	Write a piece of code to manipulate the CamelContext.
Dependencies	Click **[+]** to add the library or libraries that are required by the CamelContext or Typeconverter Registry to the Studio. Click **[...]** in the **Lib Path** field to show the **Select Module** dialog box. Select the inner module from the list or browse to the external module of your choice and click **OK** to close the dialog box. The version of the external module is extracted, If not, a default Snapshot version `0.0.1-SNAPSHOT` is given to the library. You can

	change it as needed. If you are connected to a remote project with Talend Studio, when saving the Route, the external libraries will be uploaded to the Talend Artifact Repository configured in the Talend Administration Center. The libraries with the `-SNAPSHOT` suffix in the version number will be published to the Snapshot repository. Otherwise it will be published to the Release repository. If you change the version of the library and save the Route, a new version of the library will be uploaded to the Talend Artifact Repository. For more information about how to install Talend Artifact Repository, see Talend Installation Guide. For how to set the Artifact repository connection preferences, see Talend Administration Center User Guide. For how to connect to a remote project, see Talend Studio User Guide.

Advanced settings

Use MDC Logging	Select this check box to enable the use of MDC logging. To be able to use this option, you need to open the **Log4j** view in the **[Project Settings]** dialog box by clicking **File** > **Edit Project Properties** on the toolbar of the Studio main window. Select the **Activate log4j in components** check box to activate the log4j feature. Then change the `ConversionPattern` in the log4j template to add your custom MDC property like `%X{mdcPropertyName}`. To show the MDC information in the log, use the **cLog** component and a

	cProcessor before and after the **cLog**. In the **cProcessor** before the **cLog**, enter the code `org.apache.log4j.MDC.put("Key", "Value");` to add the custom property. In the **cProcessor** after the **cLog**, enter the code `org.apache.log4j.MDC.remove("Key");` to remove the MDC property. For information on how to activate log4j in components and how to customize log4j configuration, see *Talend Studio User Guide* For more information about MDC logging, see the website http://camel.apache.org/mdc-logging.html. For more information about the **cLog** component, see cLog properties on page 109.
Check	Click this button to check the version of the libraries that are added to the Route in the Talend Artifact Repository. The libraries are listed in the table below, with the versions shown in the **Update To** column. If the version of the local library is the same as in the Talend Artifact Repository, a **#** is shown in the **Status** column. If the version of the local library is older than in the Talend Artifact Repository, a × is shown in the **Status** column, and the check box in the **Sync** column is selected by default.
Sync	Select the check box in the **Sync** column of the table for the libraries that you want to update and click the **Sync** button to download and install the newer version from the Talend Artifact Repository into the Studio.

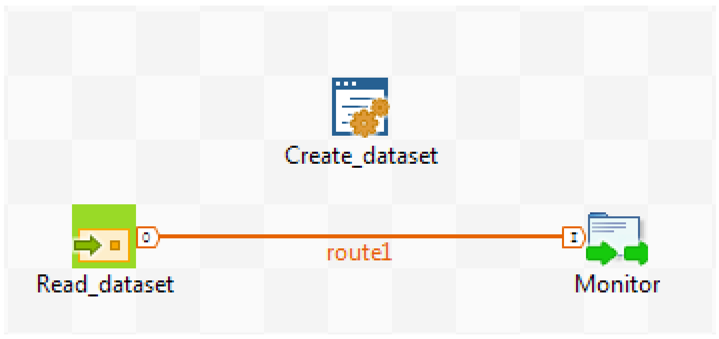
Usage

Usage rule	**cConfig** cannot be added directly in a Route.
Limitation	n/a

Scenario: Implementing a dataset from the Registry

This scenario applies only to a Talend solution with ESB.

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

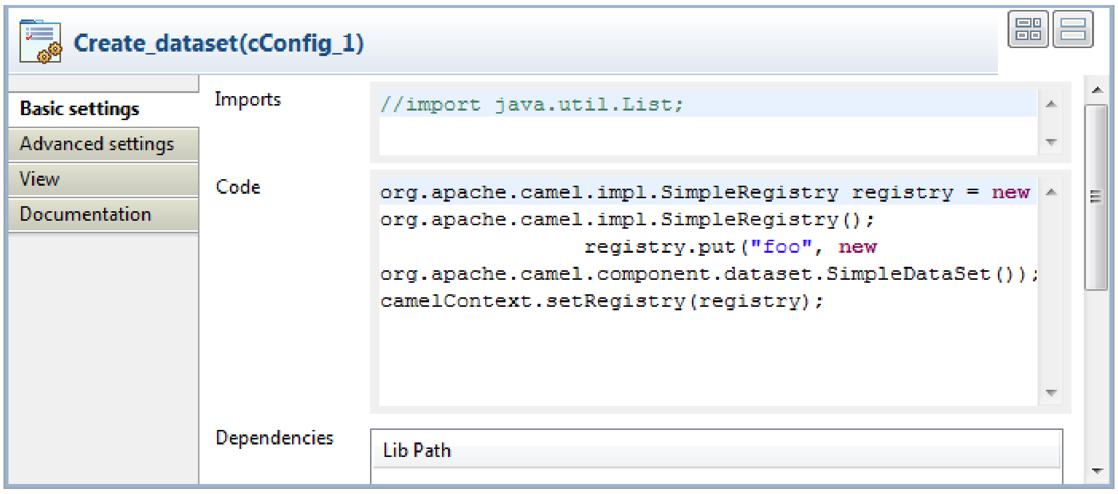


Dropping and linking the components

1. From the **Palette**, expand the **Custom** folder, and drop a **cConfig** component onto the design workspace.
2. Expand the **Connectivity** folder, and drop a **cMessagingEndpoint** component onto the design workspace.
3. Expand the **Custom** 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.


Create_dataset(cConfig_1)

Basic settings
Advanced settings
View
Documentation

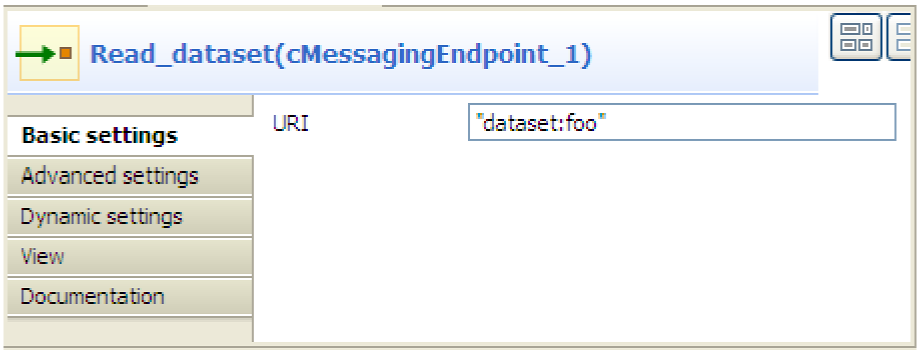
Imports
```
//import java.util.List;
```
Code
```
org.apache.camel.impl.SimpleRegistry registry = new
org.apache.camel.impl.SimpleRegistry();
                registry.put("foo", new
org.apache.camel.component.dataset.SimpleDataSet());
camelContext.setRegistry(registry);
```
Dependencies

Lib Path

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

```
SimpleDataSet dataset = new SimpleDataSet(1);
String messageBody = "testbody";
dataset.setDefaultBody(messageBody);
registry.put("foo", dataset);
```

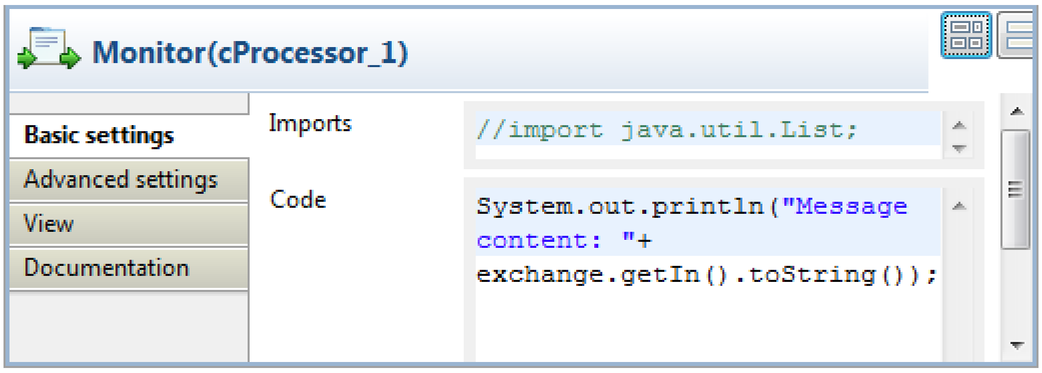
3. Double-click the input **cMessagingEndpoint** component, which is labelled *Read_dataset*, to display its **Basic settings** view in the **Component** tab.


Read_dataset(cMessagingEndpoint_1)

Basic settings
Advanced settings
Dynamic settings
View
Documentation

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.


Monitor(cProcessor_1)

Basic settings
Advanced settings
View
Documentation

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

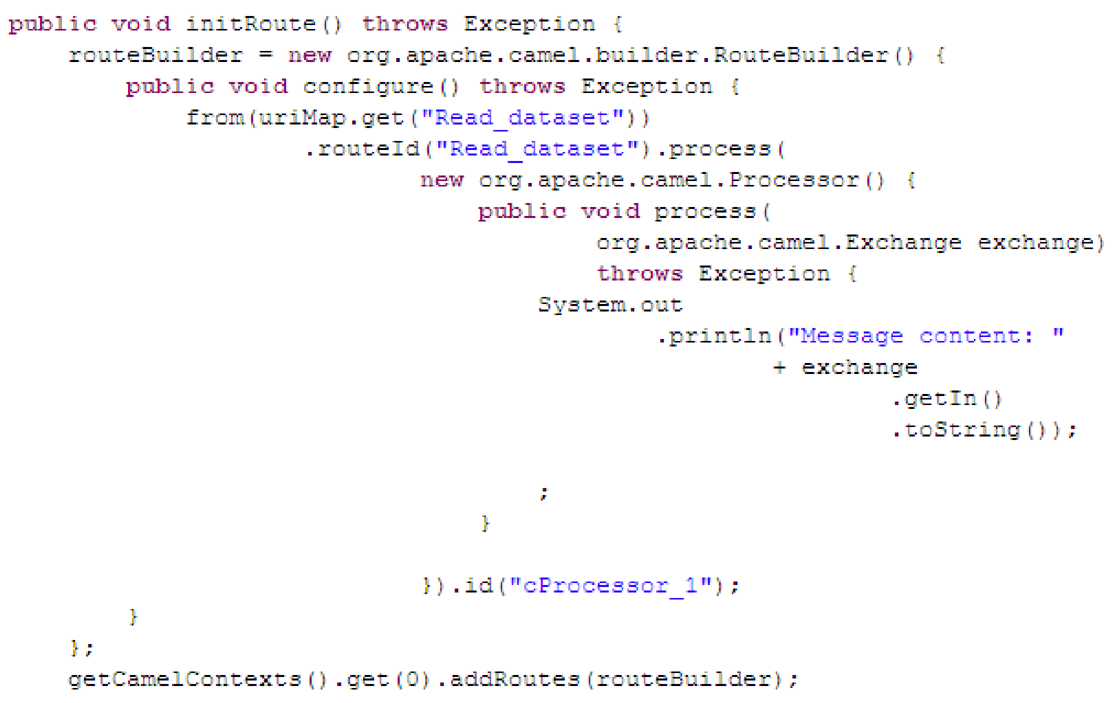
   System.out.println("Message content: "+

   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.

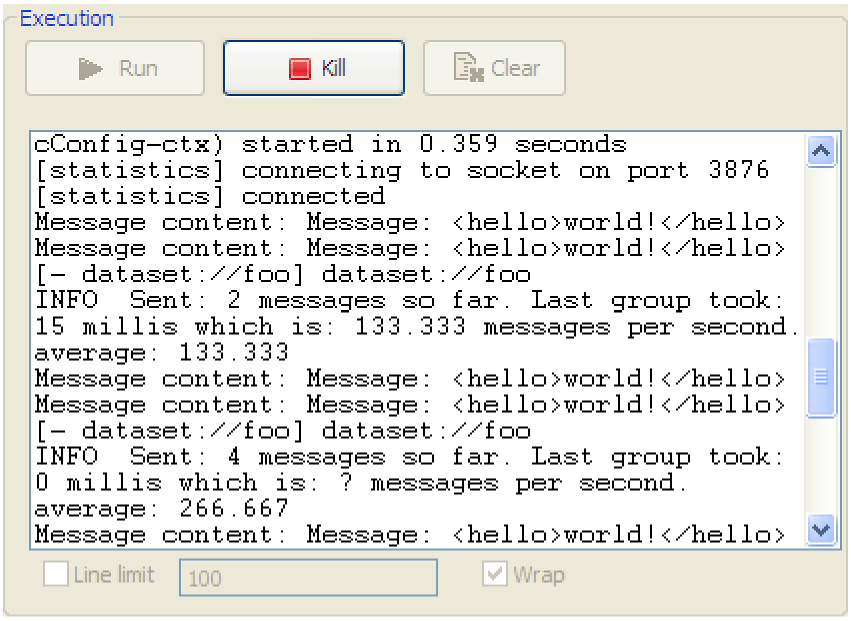


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Read_dataset"))
                    .routeId("Read_dataset").process(
                            new org.apache.camel.Processor() {
                                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>
```

cContentEnricher properties

cContentEnricher allows you to enrich the message.

cContentEnricher is designed to use a consumer or producer to obtain additional data, respectively intended for event messaging and request/reply messaging.

cContentEnricher Standard properties

These properties are used to configure cContentEnricher running in the Standard Job framework.

The Standard cContentEnricher component belongs to the Transformation family.

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, that is to say sending a message to the defined URI.
Using a consumer	Select this check box to use a consumer to obtain additional data, that is to say 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

Usage rule	**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

Scenario: Receiving messages from a list of URLs

This scenario applies only to a Talend solution with ESB.

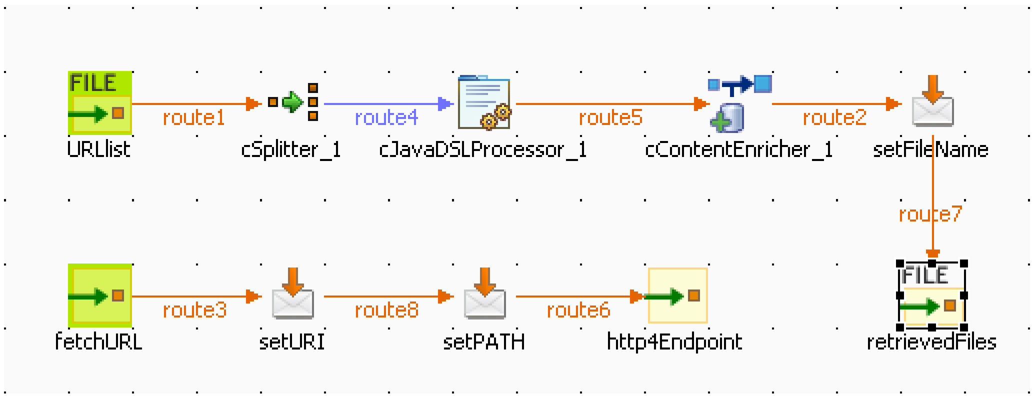
In this scenario, we will use the Camel component HTTP4 and the **cContentEnricher** component to retrieve messages from a list of URLs. To do this, we need to build two sub-routes, one to read a file with a list of URLs and send the messages to the local file system, the other to retrieve the messages on these URLs.

In this use case, we will take a list of URLs on the local Tomcat server as the example. So we need to start Apache Tomcat before executing the Route.

A TXT file `URLlist` is used to provide the list of URLs, as shown below.

docs/introduction.html

docs/setup.html



Dropping and linking the components

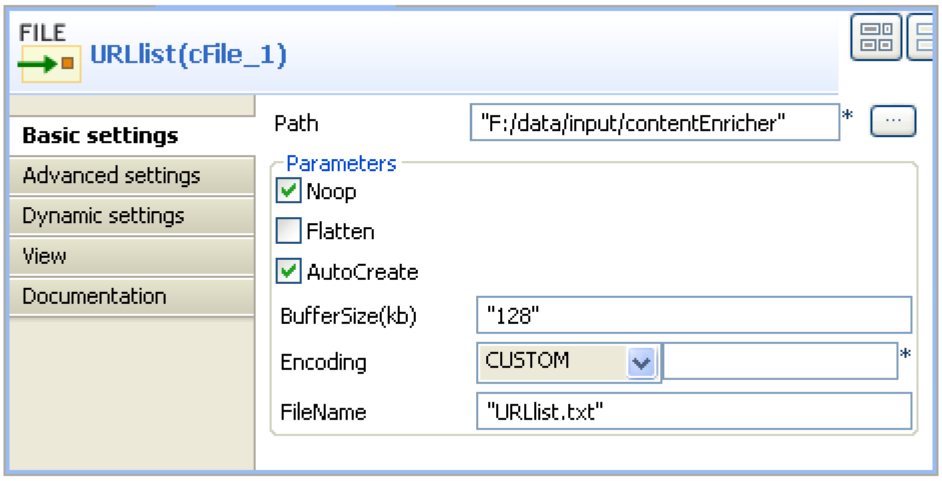
1. From the **Palette**, drag and drop a **cSplitter**, a **cJavaDSLProcessor**, a **cContentEnricher**, two **cFile**, two **cMessagingEndpoint**, and three **cSetHeader** components onto the design workspace.
2. Label the components properly for better identification of their roles and link them using the **Row** > **Route** connection as shown above.

Configuring the components

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

Configuring the first sub-route

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


URLlist(cFile_1)

Basic settings | Advanced settings | Dynamic settings | View | Documentation

Path: "F:/data/input/contentEnricher"

Parameters
- [✓] Noop
- [ ] Flatten
- [✓] AutoCreate

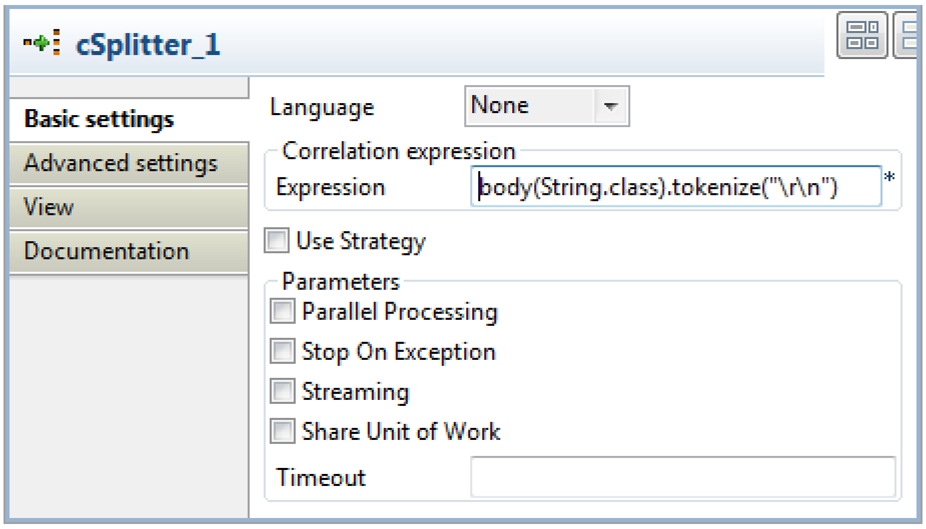
BufferSize(kb): "128"

Encoding: CUSTOM

FileName: "URLlist.txt"

2. In the **Path** field, browse to the file path where the URL list file is saved.

   In the **FileName** field, enter the filename *URLlist.txt*.
3. Double-click the **cSplitter** component to display its **Basic settings** view in the **Component** tab.



Language: None

Correlation expression

Expression: body(String.class).tokenize("\r\n")

- [ ] Use Strategy

Parameters

- [ ] Parallel Processing
- [ ] Stop On Exception
- [ ] Streaming
- [ ] Share Unit of Work

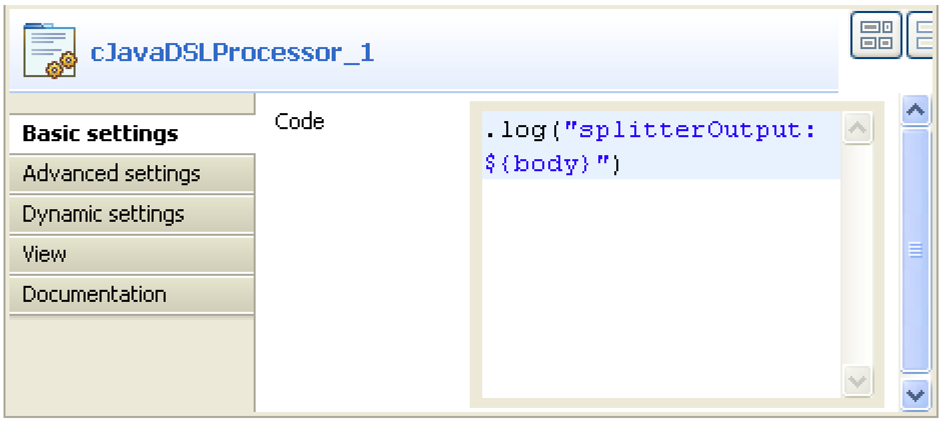
Timeout:

4. Select **None** in the **Language** list. In the **Expression** field, enter the code `body(String.class).tokenize("\r\n")` to split the message in each row into sub-messages.

   Note:

   Note that this piece of code is for Windows only. For Unix, change it to `body(String.class).tokenize("\n")`, and for Mac, `body(String.class).tokenize("\r")`.

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

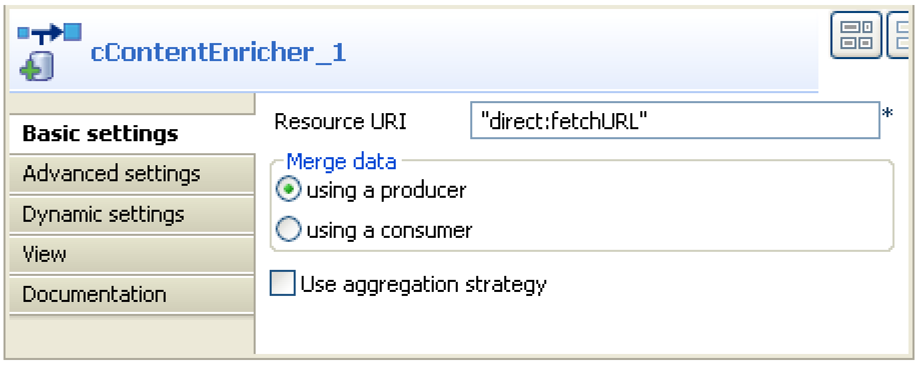


Code

```
.log("splitterOutput:
${body}")
```

6. In the **Code** area, enter the code `.log("splitterOutput: ${body}")` to get the split message body.

7. Double-click the **cContentEnricher** component display its **Basic settings** view in the **Component** tab.


cContentEnricher_1

Basic settings | Advanced settings | Dynamic settings | View | Documentation

Resource URI: "direct:fetchURL"

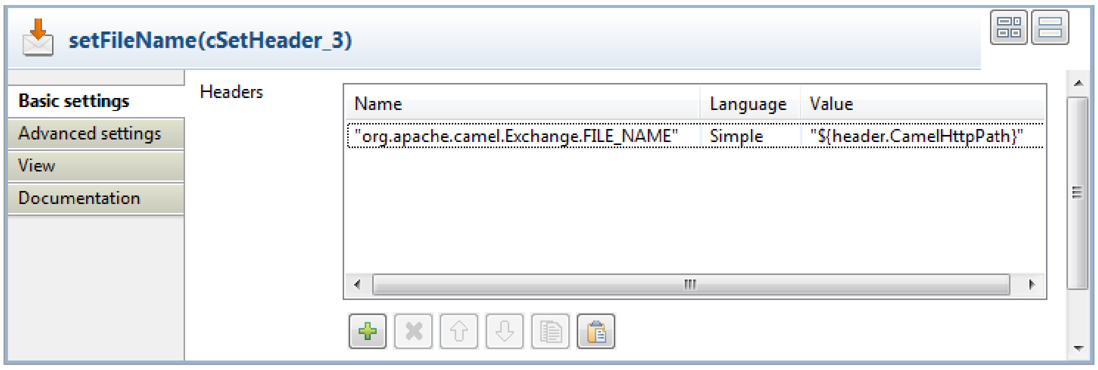
Merge data: [✓ using a producer] using a consumer

☐ Use aggregation strategy

8. Select **using a producer** to use a producer to provide additional data and send the message to a defined URI.

   In the **Resource URI** field, enter `"direct:fetchURL"` where the message will be delivered.

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


setFileName(cSetHeader_3)

Basic settings | Advanced settings | View | Documentation

Headers

Name	Language	Value
"org.apache.camel.Exchange.FILE_NAME"	Simple	"${header.CamelHttpPath}"

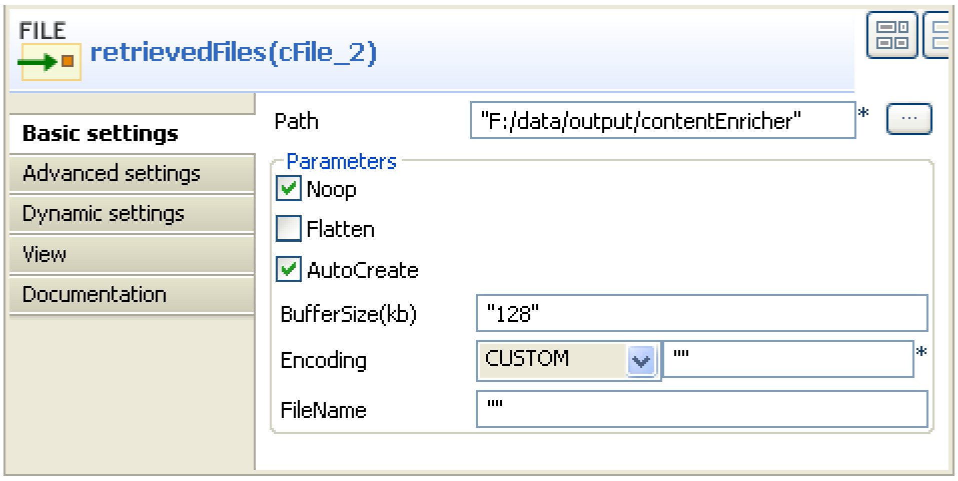
10. Click **[+]** to add a row to the **Headers** table.

    In the **Name** field, enter `org.apache.camel.Exchange.FILE_NAME` to define the file name for each incoming message.

    Select **Simple** in the **Language** list.

    In the **Value** field, enter `"${header.CamelHttpPath}"` to get the URI's path of the incoming message.

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


retrievedFiles(cFile_2)

Basic settings | Advanced settings | Dynamic settings | View | Documentation

Path: "F:/data/output/contentEnricher"

Parameters

☑ Noop

☐ Flatten

☑ AutoCreate

BufferSize(kb): "128"

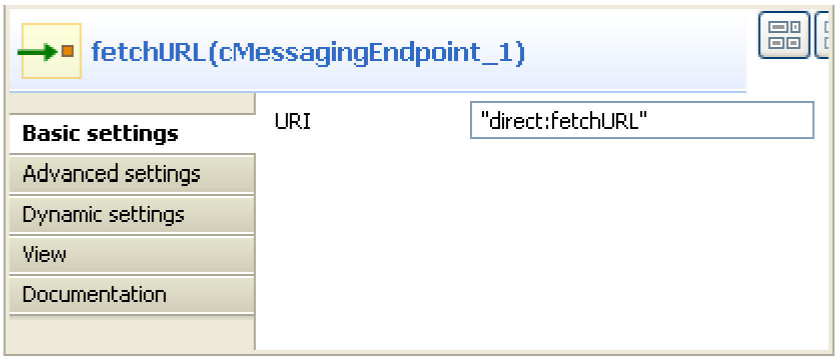
Encoding: CUSTOM ""

FileName: ""

12. In the **Path** field, browse to the destination file path where you want the messages to be saved.

Configuring the second sub-route

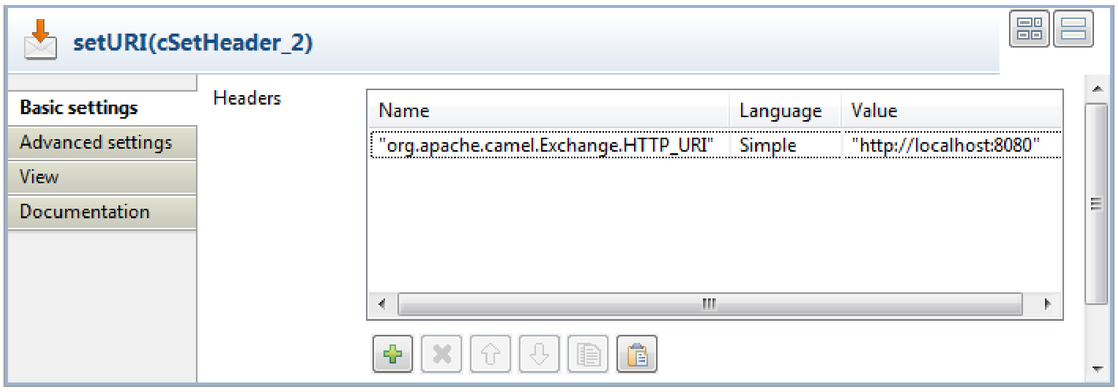
1. Double-click the **fetchURL** component to display its **Basic settings** view in the **Component** tab.



fetchURL(cMessagingEndpoint_1)

URI: "direct:fetchURL"

2. In the **URI** field, enter *"direct:fetchURL"* that is defined in the **cContentEnricher** component.
3. Double-click the **setURI** component to display its **Basic settings** view in the **Component** tab.



setURI(cSetHeader_2)

Headers

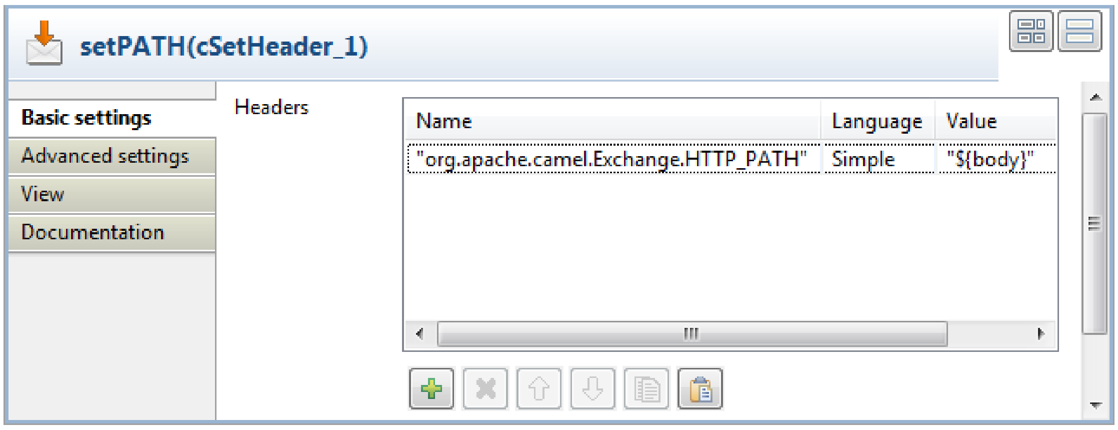
Name	Language	Value
"org.apache.camel.Exchange.HTTP_URI"	Simple	"http://localhost:8080"

4. Click **[+]** to add a row to the **Headers** table.

   In the **Name** field, enter `org.apache.camel.Exchange.HTTP_URI` to define the HTTP URI of each message.

   Select **Simple** in the **Language** list.

   In the **Value** field, enter `"http://localhost:8080"` of the local Tomcat server.
5. Double-click the **setPATH** component to display its **Basic settings** view in the **Component** tab.



setPATH(cSetHeader_1)

Headers

Name	Language	Value
"org.apache.camel.Exchange.HTTP_PATH"	Simple	"${body}"

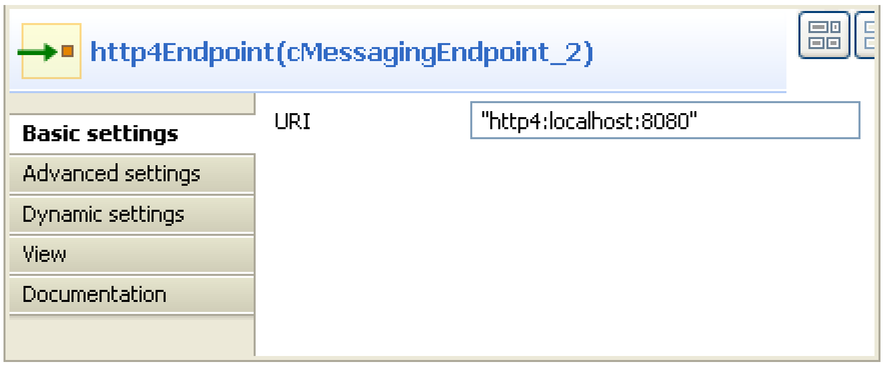
6. Click **[+]** to add a row to the **Headers** table.

   In the **Name** field, enter `org.apache.camel.Exchange.HTTP_PATH` to define the HTTP path of each message.

Select **Simple** in the **Language** list.

In the **Value** field, enter `"${body}"` that is split from the original message.

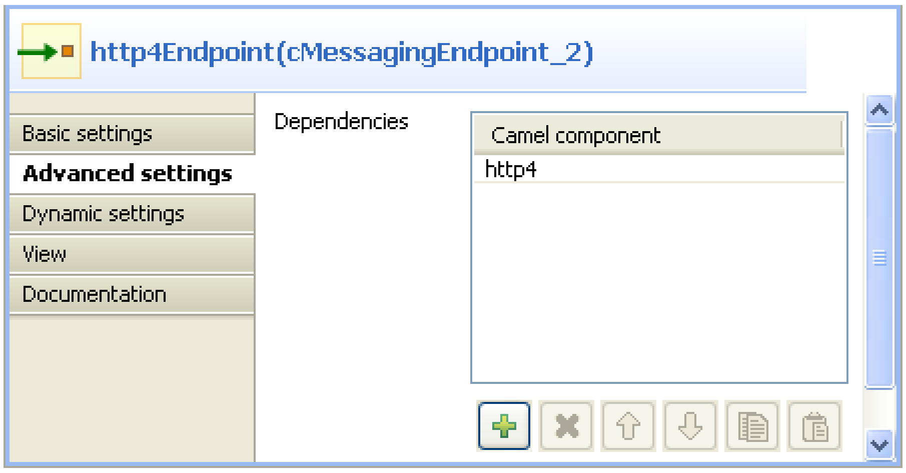
7. Double-click the **http4Endpoint** component to display its **Basic settings** view in the **Component** tab.



URI "http4:localhost:8080"

8. In the **URI** field, enter *"http4:localhost:8080"* to consuming HTTP resources on the local Tomcat server.

9. Click the **Advanced settings** view. Click ✚ at the bottom of the **Dependencies** list to add a row and select `http4` from the drop-down list. For more information about HTTP4, see the site http://camel.apache.org/http4.html.



Dependencies

Camel component
http4

Viewing code and executing the Route

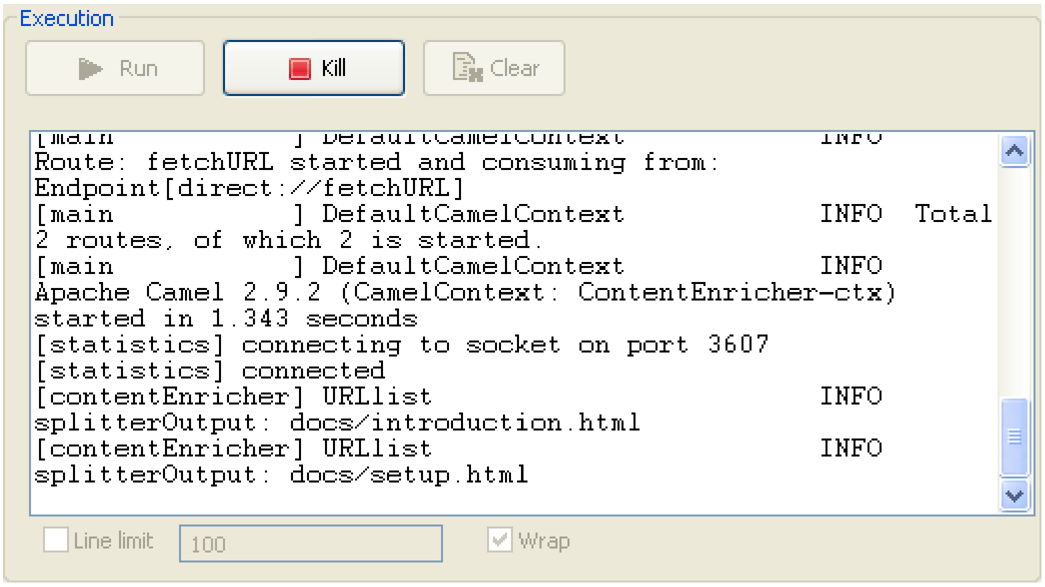
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

```
public void configure() throws java.lang.Exception {
    from(uriMap.get("URLlist_cFile_1")).routeId("URLlist_cFile_1")
            .split(body(String.class).tokenize("\r\n")).id("cSplitter_1")

            .log("splitterOutput: ${body}").id("cJavaDSLProcessor_1")
            .enrich("direct:fetchURL")

            .id("cContentEnricher_1")
            .setHeader("org.apache.camel.Exchange.FILE_NAME")
            .simple("${header.CamelHttpPath}").id("cSetHeader_1")
            .to(uriMap.get("retrievedFiles_cFile_2")).id("cFile_2");
    from(uriMap.get("fetchURL_cMessagingEndpoint_1"))
            .routeId("fetchURL_cMessagingEndpoint_1")
            .setHeader("org.apache.camel.Exchange.HTTP_URI")
            .simple("http://localhost:8080").id("cSetHeader_2")
            .setHeader("org.apache.camel.Exchange.HTTP_PATH")
            .simple("${body}").id("cSetHeader_3")
            .to(uriMap.get("http4Endpoint_cMessagingEndpoint_2"))
            .id("cMessagingEndpoint_2");
}
```

As shown above, a message route is built from the `URLlist` to the `retrievedFiles` via the `.split`, `.log`, `.enrich`, and `.setHeader`. The other message route is built from `fetchURL` to `http4Endpoint` via two `.setHeader`.

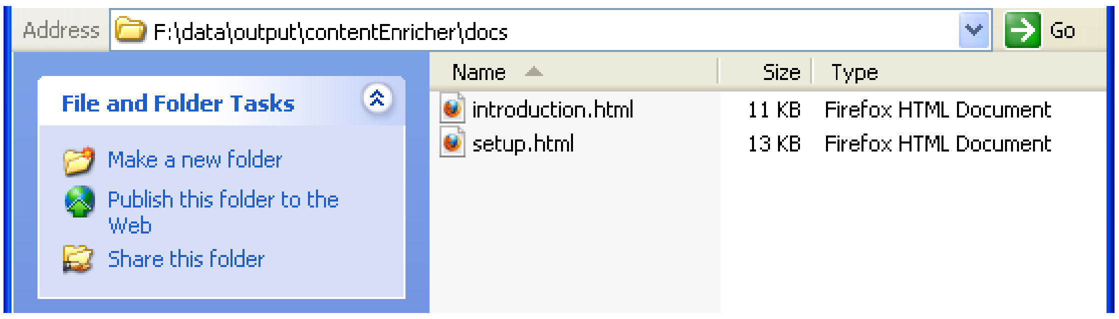
2. Press **F6** to execute the Route.

   RESULT: The split message is printed on the **Run** console.



```
[main           ] DefaultCamelContext           INFO
Route: fetchURL started and consuming from:
Endpoint[direct://fetchURL]
[main           ] DefaultCamelContext           INFO  Total
2 routes, of which 2 is started.
[main           ] DefaultCamelContext           INFO
Apache Camel 2.9.2 (CamelContext: ContentEnricher-ctx)
started in 1.343 seconds
[statistics] connecting to socket on port 3607
[statistics] connected
[contentEnricher] URLlist                       INFO
splitterOutput: docs/introduction.html
[contentEnricher] URLlist                       INFO
splitterOutput: docs/setup.html
```

The messages from the list of URLs are saved in defined directory of the local file system.


Address
F:\data\output\contentEnricher\docs
Go
File and Folder Tasks
Make a new folder
Publish this folder to the Web
Share this folder

Name	Size	Type
introduction.html	11 KB	Firefox HTML Document
setup.html	13 KB	Firefox HTML Document

cConvertBodyTo properties

cConvertBodyTo is used to change the class type of the message body.

cConvertBodyTo converts the message body to the given class type.

cConvertBodyTo Standard properties

These properties are used to configure cConvertBodyTo running in the Standard Job framework.

The Standard cConvertBodyTo component belongs to the Transformation family.

Basic settings

Target Class Name	Enter the name of the class type that you want to convert the message body to.

Usage

Usage rule	**cConvertBodyTo** is used as a middle component in a Route.
Limitation	n/a

cSOAP properties

cSOAP is used to provide or consume Web services.

cSOAP provides integration with Apache CXF for connecting to JAX-WS services.

cSOAP Standard properties

These properties are used to configure cSOAP running in the Standard Job framework.

The Standard cSOAP component belongs to the Connectivity/Services family.

Basic settings

Service/Address	The service endpoint URL where the Web service is provided. In case **cSOAP** is used to consume a Web service and the endpoint lookup shall use the Service Locator (the **Use Service Locator** check box is selected), the URL needs to be `"locator://anyAddress/"`.
Service/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. Choose **Repository** or **File** to provide the Web service from a Route Resource or the file system. **serviceClass**: Select this type to provide the Web service from an SEI (Service Endpoint Interface) Java class.
Service/WSDL File	This field appears when the **wsdlURL** service type is selected. If the WSDL file is from the file system, browse to or enter the path to the WSDL file. If the WSDL file is from a Route Resource, click **[...]** and select the one you want from the Resources tree view. The **Version** list appears allowing you to choose from all the versions of the Route Resource.
Service/Service configuration	This option appears when **wsdlURL** is selected in the **Type** list. It allows you to configure the service endpoint information conveniently. Click **[...]** to open the service configuration wizard. The **WSDL** field in the wizard is filled in with the WSDL file defined in the **WSDL File** field automatically. You can also set the WSDL file

	directly in the service configuration wizard in one of the following ways: • Click **Browse...** to browse to or enter the path to the WSDL file in the file system. • Click **Services** to select a service under the **Services**. • Click **Resources** to select a service under the **Resources** node. After setting the WSDL file, click to show the port(s) and operation(s) available in the **Port Name** and **Operation** boxes respectively. Select the one you want to use and click **Finish**. The **Operation** box only shows when the **cSOAP** component is used to consume a Web service.
Service/Service Class	This field appears when the **serviceClass** service type is selected. Enter the name of the service class to be used to provide the Web service.
Service/Dataformat	The exchange data style. **POJO**, **PAYLOAD**, **RAW**, or **CXF_MESSAGE**. **POJO**s (Plain Old Java Objects) are the Java parameters to the method being invoked on the target server. **PAYLOAD** is the message payload, the contents of the `soap:body`. **RAW** is the raw message that is received from the transport layer without SAM (Service Activity Monitor) support. **CXF_MESSAGE** is the raw message that is received from the transport layer with SAM support.
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. This field gets filled in automatically upon completion of the **Service configuration**.
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. This field gets filled in automatically upon completion of the **Service configuration**.
Allow Streaming	This option appears when **PAYLOAD** is selected in the **Dataformat** list. Select this check box to keep the payload as a `javax.xml.transform.Source` object that would allow streaming over buffering. If this option is disabled, the **cSOAP** component will parse the incoming messages into DOM elements.
Operation Name	The operation name this service is implementing. It maps to the `wsdl:operation@name`, in the format of `ns:OPERATION_NAME` where `ns` is a namespace prefix valid at this scope. This option appears when the **cSOAP** component is used consume a Web service. This field gets filled in automatically upon completion of the **Service configuration**.
Use Service Registry	This option is only available if you subscribed to Talend Enterprise ESB solutions. Select this check box to enable the service registry. It provides dynamic endpoint lookup and allows services to be redirected based upon information retrieved from the registry. It works in Runtime only.

	When the **cSOAP** component is used to provide a Web service, the service deployed in Runtime will work with the service registry. When the **cSOAP** component is used to consume a Web service: In the **Correlation Value** field, specify a correlation ID or leave this field empty. For more information, see the **Use Business Correlation** option. In the **Username** and the **Password** fields, enter the authentication credentials. To enter the password, click the **[...]** button next to the password field, and then in the pop-up dialog box enter the password between double quotes and click **OK** to save the settings. If SAML token is registered in the service registry, you need to specify the client's role in the **Role** field. You can also select the **Propagate Credentials** check box to make the call on behalf of an already authenticated user by propagating the existing credentials. You can enter the username and the password to authenticate with STS to propagate credentials using username and password, or provide the alias, username and the password to propagate using certificate. For more information, see the **Use Authentication** option. For more information about how to set up and use the Service Registry, see the *Talend Administration Center User Guide* and *Talend ESB Infrastructure Services Configuration Guide*.
Use Service Locator	Provides service consumers with a mechanism to discover service endpoints at runtime without specifying the physical location of the endpoint. Additionally, it allows service providers to

	automatically register and unregister their service endpoints at the Service Locator. **Note:** For service consumers, the URL additionally needs to be set to `"locator://anyAddress/"` in the **CXF Configuration / Address** field. 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 Infrastructure Services Configuration Guide* for how to install and configure the Service Locator.
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. This option is also disabled when the **Use Service Registry** check box is selected if you subscribed to Talend Enterprise ESB solutions.
Use Authentication	Select this check box to enable the authentication option. Select from **Username Token**, **SAML Token (ESB runtime only)**, **HTTP Basic**, and **HTTP Digest**.

	When the **cSOAP** component is used to produce a Web service, authentication with the **Username Token**, **SAML token**, and **HTTP Basic** work in runtime only. **HTTP Digest** is not supported. When **SAML Token (ESB runtime only)** is selected, **cSOAP** will get the SAML Token from the request header for further use in the message routing. When the **cSOAP** component is used to consume a Web service, authentication with the **Username Token**, **HTTP Basic**, and **HTTP Digest** work in both the studio and runtime. Authentication with the **SAML token** works in runtime only. Enter a username and a password in the corresponding fields as required. To enter the password, click the **[...]** button next to the password field, and then in the pop-up dialog box enter the password between double quotes and click **OK** to save the settings. When **SAML Token (ESB runtime only)** is selected, you can either provide the user credentials to send the request or make the call on behalf of an already authenticated user by propagating the existing credentials. Select from: **-**: Enter the username and the password in the corresponding fields to access the service. **Propagate using U/P**: Enter the user name and the password used to authenticate via STS. **Propagate using Certificate**: Enter the alias and the password used to authenticate via STS. This check box disappears when the **Use Service Registry** check box is selected.
Use Authorization	This option is only available if you subscribed to Talend Enterprise ESB solutions. It appears when

	SAML Token (ESB runtime only) is selected in the **Use Authentication** list. When the **cSOAP** component is used to provide a Web service, select this check box to enable authorization. When the **cSOAP** component is used to consume a Web service, select this check box to invoke authorized call and specify the client's role in the **Role** field. For more information about the management of user roles and rights, see the *Talend Administration Center User Guide* and *Talend ESB Infrastructure Services Configuration Guide*.
Use Business Correlation	Select this check box to enable the correlation option so that chained service calls will be grouped under the same correlation ID. When **cSOAP** is used to provide a Web service: **cSOAP** will extract the correlation ID from the request message. If the request message does not contain a correlation ID, the provider will create a correlation ID automatically in the SOAP header. When **cSOAP** is used to consume a Web service: You can specify a correlation ID in the **Correlation Value** field using a string or a simple expression. If you leave this field empty, this value will be generated automatically at runtime. The correlation ID will be created in the custom SOAP header of the request message and passed on to the service it calls. This check box disappears when the **Use Service Registry** check box is selected or **RAW** is selected in the **Dataformat** list.

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

Usage rule	**cSOAP** can be a start, middle or end component in a Route.
Limitation	Due to license incompatibility, one or more JARs required to use this component are not provided. You can install the missing JARs for this particular component by clicking the **Install** button on the **Component** tab view. You can also find out and add all missing JARs easily on the **Modules** tab in the **Integration** perspective of your studio. You can find more details about how to install external modules in Talend Help Center (https://help.talend.com). Multiple **cSOAP** components with the same label in a Route is not supported. When **cSOAP** is used to consume a Web service, if you use the **CXF_MESSAGE** data format, the request body type need to be `javax.xml.transform.Source.class`, or the request body will be empty. For simple proxy use cases, for example, from **cSOAP** to **cProcessor** to **cSOAP**, if you use the **RAW** data format, the request body will be reset. If it is printed by **cProcessor**, the output request body will be empty.

	When **cSOAP** is used to consume a Web service and the data format is **POJO**, **PAYLOAD**, or **CXF_MESSAGE**, if fault response is returned the message routing will stop. In this case, it is recommended to use the **cErrorHandler** component to catch the fault message. For more information about **cErrorHandler**, see cErrorHandler properties on page 79.

cREST properties

cREST is used to provide or consume REST-ful Web services.

cREST provides integration with Apache CXF for connecting to JAX-RS services hosted in CXF.

cREST Standard properties

These properties are used to configure cREST running in the Standard Job framework.

The Standard cREST component belongs to the Connectivity/Services family.

Basic settings

Endpoint	The service endpoint URL where the REST service is provided.
Type	Select which type you want to use to provide the REST service. Either **Manual** or **resourceClass**. **Manual**: Determine the REST API mapping manually in the table if **cREST** is used as a service provider, or set HTTP Method and other parameters if the component is used as a service consumer. **resourceClass**: Select this type to provide the resource class which you want to export as the REST service.
REST API Mapping	This table appears when the **Manual** service type is selected and **cREST** is used as a service provider.

	Click **[+]** under the table to add as many rows as needed to specify the HTTP request: **Output Flow**: Specify the name of an output flow. **HTTP Verb**: Select a HTTP method from **GET**, **POST**, **PUT**, **DELETE**, **OPTIONS** and **HEAD** in the list. **URI pattern**: Fill this field with the REST URI that describes the resource. **Consumes**: Select the format type of the consume content that the component will use from **XML or JSON**, **XML**, **JSON**, **Form**, **Multipart**, and **Any** when **HTTP Verb** is **POST** or **PUT**. **Produces**: Select the format type of the produce content that the component will use from **XML or JSON**, **XML**, **JSON**, **HTML**, **Any** when **HTTP Verb** is **GET**, **POST**, **PUT** or **DELETE**. **Bean class**: Set the bean class when the **HTTP Verb** is **POST** or **PUT** and the consume content format is **XML or JSON**, **XML** or **JSON**.
Resource Class	This field appears when the **resourceClass** service type is selected. Enter the name of the resource class which you want to export as the REST service.
Operation	This field appears when the **resourceClass** service type is selected and **cREST** is used as the service consumer. Enter the name of the operation.
Relative Path	This field appears when the **Manual** service type is selected and **cREST** is used as the service consumer. Enter the relative path of the REST server to be invoked.

HTTP Method	This option appears when the **Manual** service type is selected and **cREST** is used as the service consumer. Select a HTTP method from**GET**, **POST**, **PUT**, and **DELETE** in the list. **Warning:** When using the **POST** method to create an object, by default, the created object ID will not be get back from the header. By default, the **cREST** filters any header except system headers. To get the created object ID, you need to set `javax.ws.rs.core.Response` as the response class.
Content Type	This option appears when the **Manual** service type is selected and **cREST** is used as the service consumer. Select **XML**, **JSON**, or **FORM** according to the media type of the content to be uploaded to the server end. This list appears only when you select the **POST** or **PUT** in the **HTTP Method** list.
Accept Type	This field appears when the **Manual** service type is selected and **cREST** is used as the service consumer. Select the media type the client end is prepared to accept for the response from the server end. Available options are **XML**, **JSON**, and **ANY**. When **ANY** is selected, the response message can be of any type and will be transformed into a string. This list does not appear when you select the **DELETE** method.

Response Class	This field appears when the **Manual** service type is selected and the **cREST** is used as the service consumer. Enter the name of the response class.
Use Service Locator	Select this check box to enable the Service Locator. Specify the service namespace and the service name in the corresponding fields.
Enable the Service Activity Monitoring	Select this check box to enable the Service Activity Monitor. Note that this option works in Runtime only. When running the Route in the Studio, it is recommended to clear this check box. Otherwise warnings will be thrown in the execution console.
Use Authentication	Select this check box to enable the authentication option. Select the authentication type from: • **HTTP Basic**: The simplest technique for enforcing access controls to web resources using standard fields in the HTTP header. • **SAML Token (ESB runtime only)**: An XML-based, open-standard data format for exchanging authentication and authorization data between an identity provider and a service provider. • **Open ID Connect**: An extension for OAuth2 which allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner. When the **cREST** component is used as consumer, enter a username and a password in the corresponding fields as required. To enter the password, click the **[...]** button next to the

	password field, and then in the pop-up dialog box enter the password between double quotes and click **OK** to save the settings.
Use Authorization	This option is only available if you subscribed to Talend Enterprise ESB solutions. It appears when **SAML Token (ESB runtime only)** is selected in the **Use Authentication** list. When the **cREST** component is used as the service provider, select this check box to enable authorization. When the **cREST** component is used as the service consumer, select this check box to invoke authorized call and specify the client's role in the **Role** field.
Use Business Correlation	Select this check box to create a correlation ID in this component. You can specify a correlation ID in the **Correlation Value** field.

Advanced settings

Log messages	Select this check box to log the message exchanges in the Route.
Expose Swagger specification	This option appears when **cREST** is used as a service provider. Select this check box to expose the Swagger specification and include the Swagger UI into the REST service, which provides an online API Documentation in human-readable form and some basic test features. If the Route is running in the Studio, the Swagger specification will be available at `http://127.0.0.1:8090/ENDPOINT/swagger.json` and

	`http://127.0.0.1:8090/ENDPOINT/swagger.yaml`. The Swagger UI is not available. If the Route is running in Talend Runtime, the Swagger specification will be available at `http://127.0.0.1:8040/services/ENDPOINT/swagger.json` and `http://127.0.0.1:8040/services/ENDPOINT/swagger.yaml`. The Swagger UI will be available at `http://127.0.0.1:8040/services/ENDPOINT/api-docs?url=/services/ENDPOINT/swagger.json`. If the Route is running as a ESB Microservice, the Swagger specification will be available at `http://127.0.0.1:8065/services/ENDPOINT/swagger.json` and `http://127.0.0.1:8065/services/ENDPOINT/swagger.yaml`. The Swagger UI will be available at `http://127.0.0.1:8065/services/ENDPOINT/api-docs?url=/services/ENDPOINT/swagger.json`. For more information about how to build a Route to a ESB Microservice and how to run the Microservice, see Talend Studio User Guide.
Include Documentation into Swagger Spec	This option appears when **cREST** is used as a service provider and the **Expose Swagger specification** check box is selected. Select this option to add content in the **Comment** field of the **Documentation** tab of this component into the Swagger specification and the Swagger UI page. For more information about the **Documentation** tab, see Talend Studio User Guide.

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/cxfrs.html for available URI options.

Usage

Usage rule	**cREST** can be a start component in a Route as the service provider, or middle or end component as the service consumer.
Limitation	Due to license incompatibility, one or more JARs required to use this component are not provided. You can install the missing JARs for this particular component by clicking the **Install** button on the **Component** tab view. You can also find out and add all missing JARs easily on the **Modules** tab in the **Integration** perspective of your studio. You can find more details about how to install external modules in Talend Help Center (https://help.talend.com).

Scenario: Providing and consuming a REST service using cREST

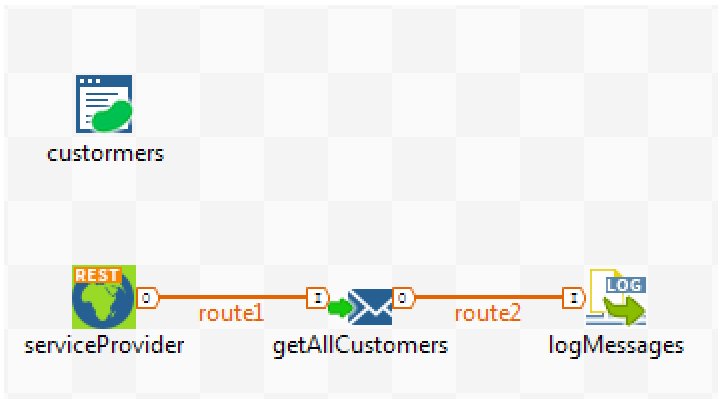
This scenario applies only to a Talend solution with ESB.

This scenario demonstrates how to use the **cREST** component to provide and consume a REST service. To do so, two Routes are built, a service provider Route and a consumer Route. The service provider Route will be accessible for requests and respond with some predefined customer information. The consumer Route will send a request to the REST service.

Building the service provider Route

This Route provides a REST Web service using the **cREST** component. In this Route, a **cBeanRegister** component is used to set the customer information in a Java bean. The bean is then called by a **cSetBody** as the response of the service.

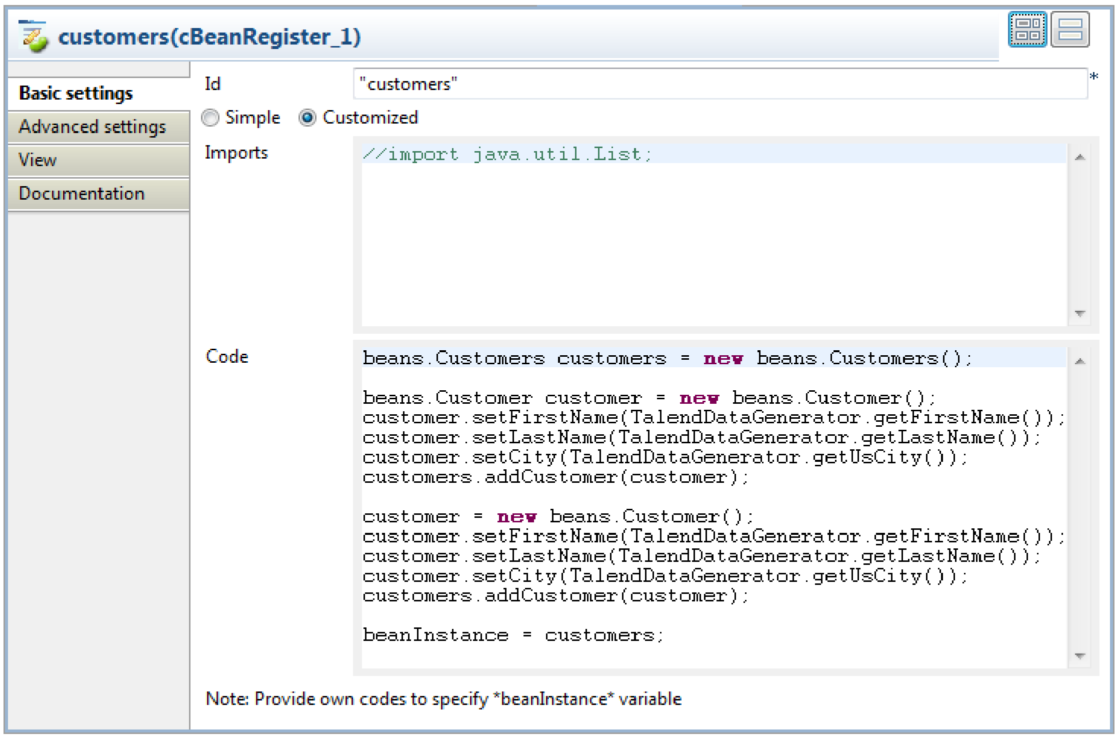
Dropping and linking the components



1. From the **Palette**, drag and drop a **cREST**, a **cSetBody**, a **cLog** and a **cBeanRegister** component onto the design workspace.
2. Link the **cREST**, **cSetBody** and **cLog** using the **Row** > **Route** connection.
3. Label the components for better identification of their roles.

Configuring the components

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



Id: "customers"

Simple [✓ Customized]

Imports:

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

Code:

```
beans.Customers customers = new beans.Customers();

beans.Customer customer = new beans.Customer();
customer.setFirstName(TalendDataGenerator.getFirstName());
customer.setLastName(TalendDataGenerator.getLastName());
customer.setCity(TalendDataGenerator.getUsCity());
customers.addCustomer(customer);

customer = new beans.Customer();
customer.setFirstName(TalendDataGenerator.getFirstName());
customer.setLastName(TalendDataGenerator.getLastName());
customer.setCity(TalendDataGenerator.getUsCity());
customers.addCustomer(customer);

beanInstance = customers;
```

Note: Provide own codes to specify *beanInstance* variable

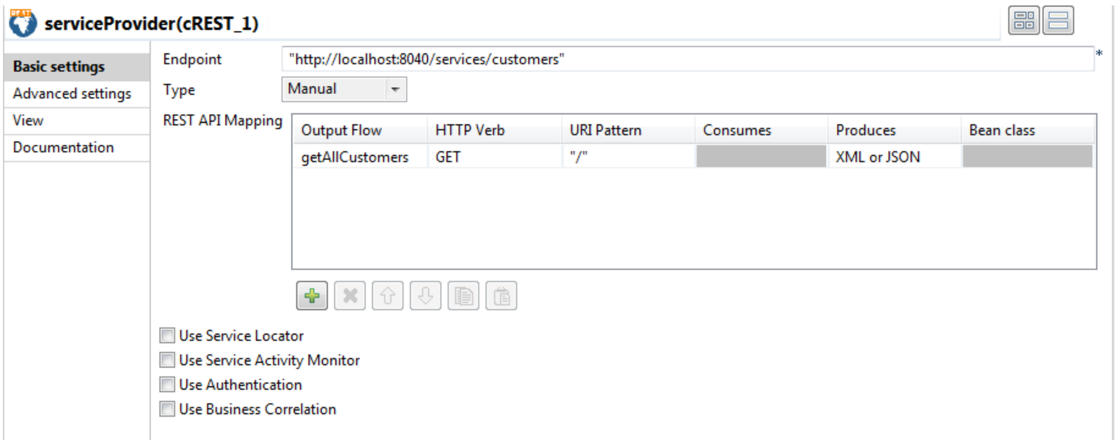
2. The **cBeanRegister** component registers a Java bean, in which the customer information is set with the `firstName`, `lastName`, `city`, and `id` values.

   In the **Id** field, enter *"customers"* to name the bean.

   Select the **Customized** option and enter the following code in the **Code** box to create two customers and set the `firstName`, `lastName`, `city`, and `id` values for each of them:

```
beans.Customers customers = new beans.Customers();
beans.Customer customer = new beans.Customer();
customer.setFirstName(TalendDataGenerator.getFirstName());
customer.setLastName(TalendDataGenerator.getLastName());
customer.setCity(TalendDataGenerator.getUsCity());
customers.addCustomer(customer);

customer = new beans.Customer();
customer.setFirstName(TalendDataGenerator.getFirstName());
customer.setLastName(TalendDataGenerator.getLastName());
customer.setCity(TalendDataGenerator.getUsCity());
customers.addCustomer(customer);

beanInstance = customers;
```

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


serviceProvider(cREST_1)

Endpoint: "http://localhost:8040/services/customers"

Type: Manual

REST API Mapping

Output Flow	HTTP Verb	URI Pattern	Consumes	Produces	Bean class
getAllCustomers	GET	"/"		XML or JSON	

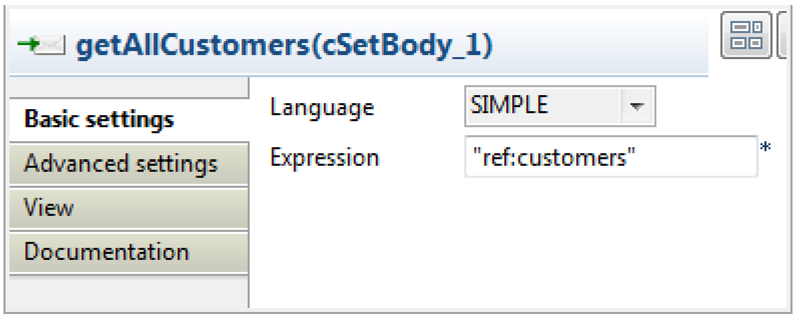
☐ Use Service Locator
☐ Use Service Activity Monitor
☐ Use Authentication
☐ Use Business Correlation

4. The **cREST** component is used to provide the REST service.

   In the **Endpoint** field, type in the endpoint URL where the Web service will be provided, *"http://localhost:8040/services/customers"* in this example.

   From the **Type** list, select **Manual** to determine the REST API mapping manually.

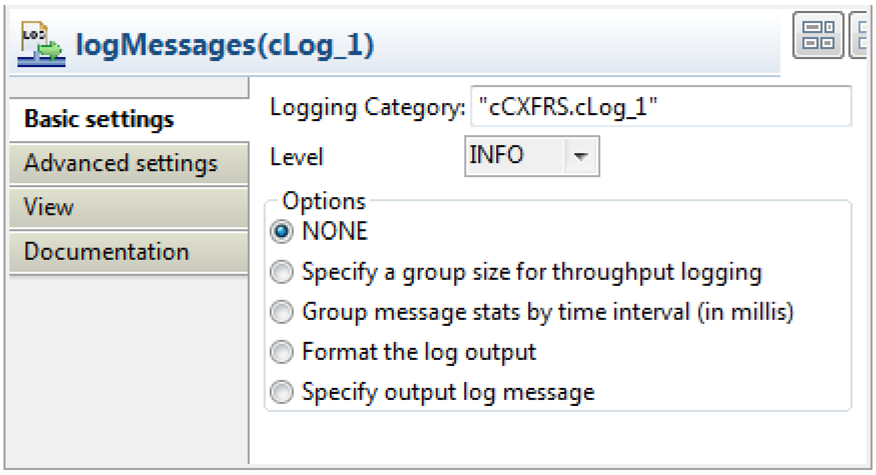
   In the **REST API mapping** table, click **[+]** to add a row in the table. In the **Output Flow** field, enter *getAllCustomers* as the name of it. Select **GET** in the **HTTP Verb** list. Keep the default settings in the other columns.
5. Double-click the **cSetBody** component to display its **Basic settings** view in the **Component** tab.


getAllCustomers(cSetBody_1)

Language: SIMPLE

Expression: "ref:customers"

6. Select **SIMPLE** from the **Dataformat** list. In the **Expression** field, enter *"ref:customers"* to refer to the bean defined in the **cBeanRegister** component as the message body of the service response.
7. Keep the default settings of the **cLog** component to log the message exchanges.


logMessages(cLog_1)

Logging Category: "cCXFRS.cLog_1"

Level: INFO

Options
- [✓ NONE]
- Specify a group size for throughput logging
- Group message stats by time interval (in millis)
- Format the log output
- Specify output log message

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

Viewing code and executing the Route

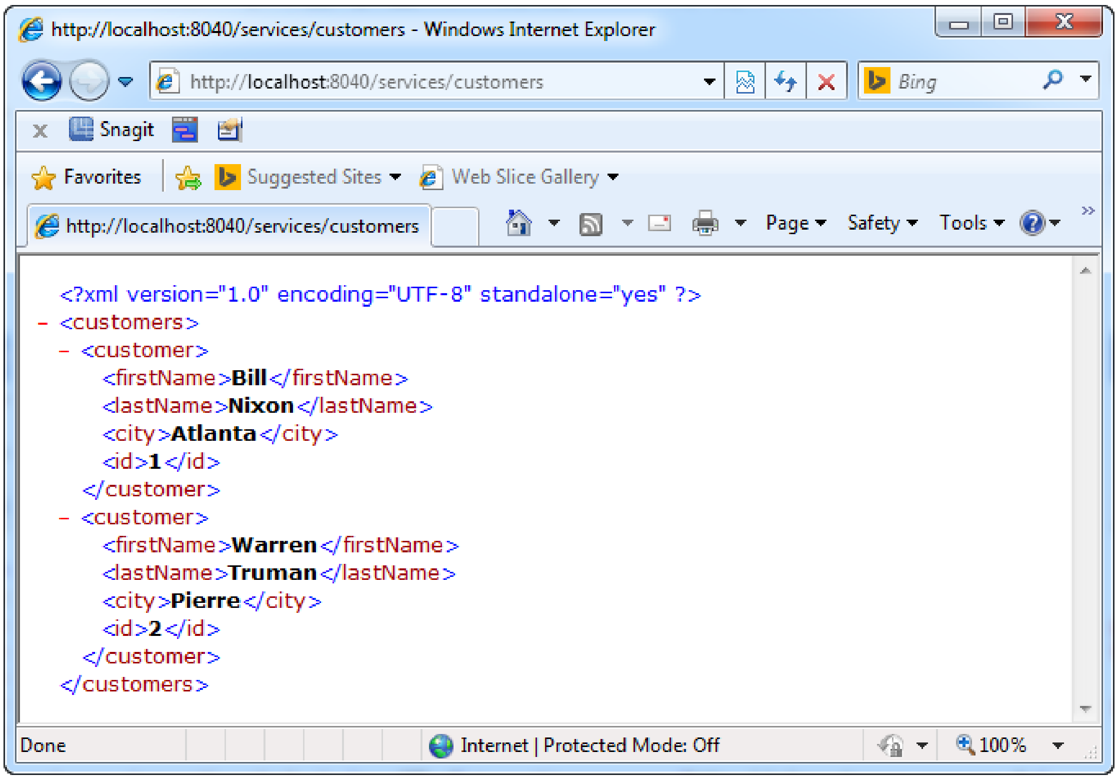
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

```
from("cxfrs://bean://cREST_1" + "?loggingFeatureEnabled=false")
        .process(new org.apache.camel.Processor() {
            public void process(org.apache.camel.Exchange exchange)
                    throws Exception {
                org.apache.camel.Message inMessage = exchange.getIn();
                inMessage
                        .setHeader(
                                "http_query",
                                org.apache.cxf.jaxrs.utils.JAXRSUtils.getStructuredParams(
                                        (String) inMessage
                                                .getHeader(org.apache.camel.Exchange.HTTP_QUERY),
                                        "&", false, false));
            }
        }).routeId("provider_cREST_1").setBody()
        .simple("ref:customers").id("provider_cSetBody_1")
        .to("log:cRest.cLog_1" + "?level=WARN")

        .id("provider_cLog_1");
```

As shown in the code, the Route is built `from cREST_1`, set message body in `cSetBody_1` and then to `cLog_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 service is successfully started. You can access it from a Web browser using the service endpoint URL. The customer information is shown in the browser.

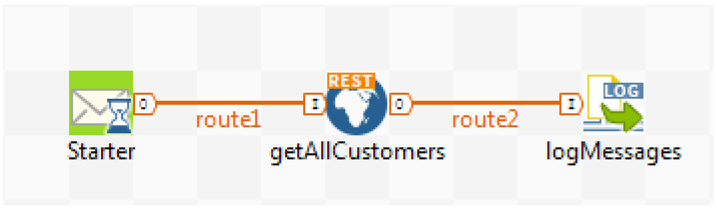

http://localhost:8040/services/customers - Windows Internet Explorer
http://localhost:8040/services/customers
Bing
Snagit
Favorites
Suggested Sites
Web Slice Gallery
http://localhost:8040/services/customers
Page
Safety
Tools
Done
Internet | Protected Mode: Off
100%

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <customers>
  - <customer>
      <firstName>Bill</firstName>
      <lastName>Nixon</lastName>
      <city>Atlanta</city>
      <id>1</id>
    </customer>
  - <customer>
      <firstName>Warren</firstName>
      <lastName>Truman</lastName>
      <city>Pierre</city>
      <id>2</id>
    </customer>
  </customers>
```

Building the service consumer Route

This Route will consume the REST service that is built in the provider Route.

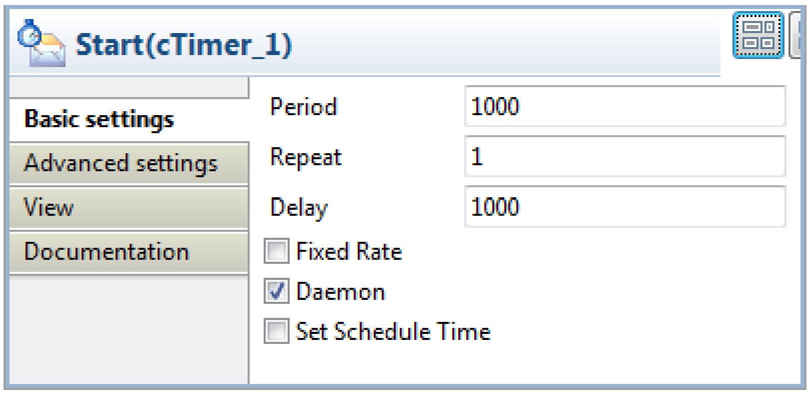
Arranging the flow of the message



1. From the **Palette**, drag and drop a **cTimer**, a **cREST**, and a **cLog** component onto the design workspace.
2. Link the **cTimer**, **cREST** and **cLog** using the **Row** > **Route** connection.
3. Label the components for better identification of their roles.

Configuring how the message is processed

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


Start(cTimer_1)

Basic settings | Advanced settings | View | Documentation

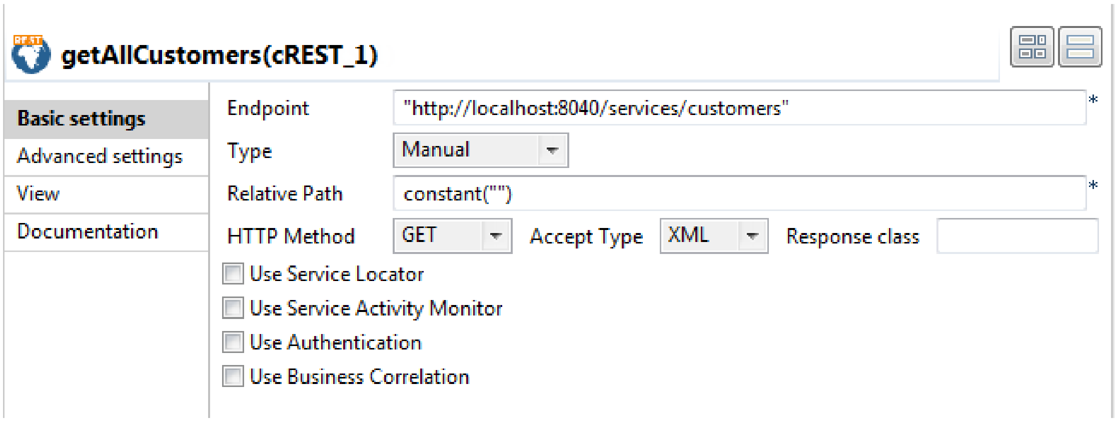
Period: 1000

Repeat: 1

Delay: 1000

- [ ] Fixed Rate
- [✓] Daemon
- [ ] Set Schedule Time

2. In the **Repeat** field, enter *1* to generate the message exchange one time. Keep the default settings of the other options.
3. Double-click the **cREST** component to display its **Basic settings** view in the **Component** tab.


getAllCustomers(cREST_1)

Basic settings | Advanced settings | View | Documentation

Endpoint: "http://localhost:8040/services/customers" *

Type: Manual

Relative Path: constant("") *

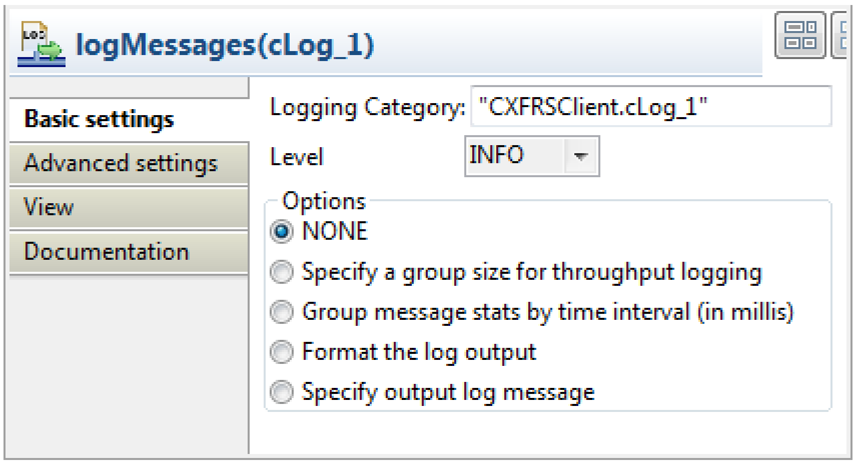
HTTP Method: GET Accept Type: XML Response class:

- [ ] Use Service Locator
- [ ] Use Service Activity Monitor
- [ ] Use Authentication
- [ ] Use Business Correlation

4. This **cREST** component will consume the REST service built in the provider Route.

   In the **Endpoint** field, type in the URL of the service, *"http://localhost:8040/services/customers"* in this example.

   Select **Manual** from the **Type** list. In the **Relative Path** field, enter *constant("")*. Select **GET** in the **HTTP Method** list. Keep the default settings of the other options.
5. Keep the default settings of the **cLog** component to log the message exchanges.


logMessages(cLog_1)

Basic settings | Advanced settings | View | Documentation

Logging Category: "CXFRSClient.cLog_1"

Level: INFO

Options
- [✓ NONE]
- Specify a group size for throughput logging
- Group message stats by time interval (in millis)
- Format the log output
- Specify output log message

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

Executing the Route

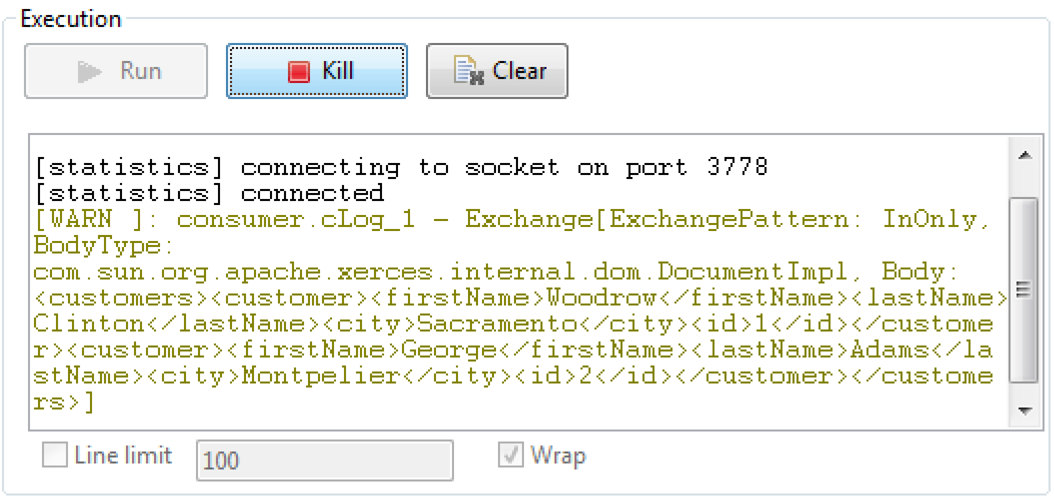
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

```
from("timer:cTimer_1" + "?repeatCount=" + 1 + "&delay=" + 1000)
        .routeId("consumer_cTimer_1")
        .setHeader(org.apache.camel.Exchange.HTTP_PATH, constant(""))
        .setHeader(org.apache.camel.Exchange.HTTP_METHOD,
                constant("GET"))
        .setHeader(org.apache.camel.Exchange.ACCEPT_CONTENT_TYPE,
                constant("application/xml"))
        .inOut("cxfrs://bean://cREST_1"
                + "?loggingFeatureEnabled=false")
        .unmarshal(new org.apache.camel.spi.DataFormat() {
            public java.lang.Object unmarshal(
                    org.apache.camel.Exchange exchange,
                    java.io.InputStream is) throws java.lang.Exception {
                java.lang.Object b = exchange.getOut().getBody();
                if (b instanceof org.apache.cxf.jaxrs.impl.ResponseImpl) {
                    org.apache.cxf.jaxrs.impl.ResponseImpl r = (org.apache.cxf.jaxrs.impl.ResponseImpl) b;
                    if ("javax.ws.rs.core.Response.class"
                            .equalsIgnoreCase("org.w3c.dom.Document.class")) {
                        return org.w3c.dom.Document.class.cast(r);
                    }
                    int status = r.getStatus();
                    if ((status < 200 || status == 204)
                            && r.getLength() <= 0 || status >= 300) {
                        return null;
                    }
                    return r.doReadEntity(org.w3c.dom.Document.class,
                            org.w3c.dom.Document.class,
                            new java.lang.annotation.Annotation[] {});
                }
                return b;
            }

            public void marshal(org.apache.camel.Exchange exchange,
                    Object o, java.io.OutputStream os) throws Exception {
            }
        }).id("consumer_cREST_1")
        .to("log:consumer.cLog_1" + "?level=WARN")

        .id("consumer_cLog_1");
```

As shown in the code, the Route is built `from` `cTimer_1`. The `HTTP_PATH`, `HTTP_METHOD`, and `ACCEPT_CONTENT_TYPE` are set in `cREST_1`. The message is then routed to `cLog_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 customers information is displayed in the console.



```
[statistics] connecting to socket on port 3778
[statistics] connected
[WARN ]: consumer.cLog_1 - Exchange[ExchangePattern: InOnly,
BodyType:
com.sun.org.apache.xerces.internal.dom.DocumentImpl, Body:
<customers><customer><firstName>Woodrow</firstName><lastName>
Clinton</lastName><city>Sacramento</city><id>1</id></custome
r><customer><firstName>George</firstName><lastName>Adams</la
stName><city>Montpelier</city><id>2</id></customer></custome
rs>]
```

cDataset properties

The **cDataset** component provides a mechanism to easily perform load and soak testing of your system. It works by allowing you to create dataset instances both as a source of messages and as a way to assert that the data set is received.

The **cDataset** component allows you to create a new dataset or reference an existing dataset to send or receive messages.

cDataset Standard properties

These properties are used to configure cDataset running in the Standard Job framework.

The Standard cDataset component belongs to the Testing family.

Basic settings

Id	The ID of the Dataset bean.
Produce Delay	Specify a delay in milliseconds to cause producers to pause.
Consume Delay	Specify a delay in milliseconds to cause consumers to pause.
Preload Size	Specify how many messages should be sent before the Route completes its initialization.
Initial Delay	Specify the time in milliseconds to wait before starting sending messages.
Minimum Rate	Specify the least number of messages that the dataset should contain before starting sending messages.
Register new Bean	Select this check box to register a new bean.
Bean Class	Enter the class of the bean. This field appears when the **Register new Bean** check box is selected.
Arguments	Set the optional arguments in the corresponding table. Click **[+]** as many times as required to add arguments to the table. This table appears when the **Register new Bean** check box is selected.

Usage

Usage rule	**cDataset** can be a start, middle, or end component of a Route.

Limitation	n/a

cDelayer properties

The **cDelayer** component allows you to set a latency in message routing.

The **cDelayer** component delays the delivery of messages.

cDelayer Standard properties

These properties are used to configure cDelayer running in the Standard Job framework.

The Standard cDelayer component belongs to the Orchestration family.

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

Usage rule	This component is usually used in the middle of a Route.
Limitation	n/a

cDirect properties

cDirect allows you to produce and consume messages synchronously in different threads within a single CamelContext.

cDirect provides direct, synchronous invocation of any consumers when a producer sends a message exchange.

cDirect Standard properties

These properties are used to configure cDirect running in the Standard Job framework.

The Standard cDirect component belongs to the Core family.

Basic settings

Name	This option appears when **cDirect** is used as a start component in a Route.

	Type in any string that uniquely identifies the endpoint.
Use Exist cDirect	This option appears when **cDirect** is used as a middle or end component in a Route. Click **[...]** and select the corresponding consumer in the dialog box.

Usage

Usage rule	**cDirect** is used as a start, middle, or end component in a Route.
Limitation	n/a

Related scenario:

For a related scenario, see Scenario: Using cSEDA, cVM and cDirect to produce and consume messages separately on page 186.

cDirectVM properties

cDirectVM allows you to produce and consume messages synchronously in different threads within a single CamelContext and across CamelContexts in the same JVM. You can use this mechanism to communicate across Web applications.

cDirectVM provides direct, synchronous invocation of any consumers when a producer sends a message exchange. It supports communication within the same CamelContext and across CamelContexts in the same JVM.

cDirectVM Standard properties

These properties are used to configure cDirectVM running in the Standard Job framework.

The Standard cDirectVM component belongs to the Core family.

Basic settings

When using as a start component in a Route:	
Endpoint Name	Type in any string that uniquely identifies the endpoint.
When using as a middle or end component in a Route:	

Select From Existing(s)	Click this radio button to select an existing consumer. Click **[...]** beside the **Consumer** field to show the existing consumer(s) and select the one to consume the message.
Input Endpoint Name	Click this radio button to enter the name of the consumer in the **Endpoint Name** field.
Block if Consumer is not active	Select this check box to let the producer block if the consumer is not active in the **Timeout** period.
Timeout	This option appears when the block is enabled. Specify the time in milliseconds before the producer stops waiting for the consumer to become active.

Usage

Usage rule	**cDirectVM** is used as a start, middle, or end component in a Route.
Limitation	n/a

cDynamicRouter properties

cDynamicRouter is used to route a message or messages to different endpoints on specified conditions.

cDynamicRouter allows you to route messages while avoiding the dependency of the router on all possible destinations.

cDynamicRouter Standard properties

These properties are used to configure cDynamicRouter running in the Standard Job framework.

The Standard cDynamicRouter component belongs to the Routing family.

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

Usage rule	**cDynamicRouter** is used as a middle or end component in a Route.
Limitation	n/a

cErrorHandler properties

cErrorHandler is used to process errors in the message routing.

cErrorHandler provides multiple strategies to deal with errors in a Route.

cErrorHandler Standard properties

These properties are used to configure cErrorHandler running in the Standard Job framework.

The Standard cErrorHandler component belongs to the Exception Handling family.

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.
	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

Usage rule	**cErrorHandler** is used separately or as a middle or end component in a Route. If this component is used separately, it will handle errors in all sub-routes. If this component is used in the middle or end of a sub-route, it will only handle exceptions that happen in the components of this sub-route, either before or after the **cErrorHandler**.

Limitation	n/a

cExchangePattern properties

cExchangePattern allows you to set the message exchange mode.

cExchangePattern can be configured to indicate the message exchange mode.

cExchangePattern Standard properties

These properties are used to configure cExchangePattern running in the Standard Job framework.

The Standard cExchangePattern component belongs to the Core family.

Basic settings

Exchange Patterns	Select the message exchange mode from **InOnly** or **InOptionalOut**, **InOut**, **OutIn**, **OutOptionalIn**, **RobustInOnly**, **RobustOutOnly**.

Usage

Usage rule	As a middle component in a Route, **cExchangePattern** allows you to set the message exchange mode.
Limitation	

cFile properties

cFile allows files to be processed by any other Camel components or messages from other components to be saved to disk.

cFile provides access to file systems.

cFile Standard properties

These properties are used to configure cFile running in the Standard Job framework.

The Standard cFile component belongs to the Connectivity/File family.

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

Usage rule	**cFile** can be a start, middle or end component in a Route.
Limitation	n/a

cFlatPack properties

cFlatPack consumes from flatpack files to object model.

cFlatPack supports fixed width and delimited file parsing via the FlatPack library.

cFlatPack Standard properties

These properties are used to configure cFlatPack running in the Standard Job framework.

The Standard cFlatPack component belongs to the Transformation family.

Basic settings

When using as a start component in a route:	
PZMAP FileType	The PZMAP file is the Flatpack configuration file that is used to configure the structure of the input file. For more information about the PZMAP file configuration, see the website http://flatpack.sourceforge.net/documentation/index.html. Select the PZMAP file type from **Filename** and **Repository Resource**. **Filename**: The PZMAP file is stored in the local file system. **Repository Resource**: The PZMAP file is stored in the **Resources** node of the **Repository**.
PZMAP Filename	This option appears when **Filename** is selected in the **PZMAP FileType** list. Enter or browse to the path to the PZMAP file.
PZMAP Repository Resource	This option appears when **Repository Resource** is selected in the **PZMAP FileType** list. Click **[...]** and select the PZMAP file under the **Resources** node in the dialog box.
Fixed Positional file	Select this option if the file is a fixed format file.
Delimited file	Select this option if the file is delimited.

	Text Qualifier: Specify the text qualifier for delimited files. The default value is `"`. **Text Delimiter**: Specify the character delimiter for delimited files. The default value is `,`.
Split Rows	Select this check box to process each row one by one.
Ignore First Record	Select this check box to ignore the first line for delimited files (for the column headers). This option is only available if the **Delimited file** option is selected.
Allow Short Lines	Select this check box to allow lines shorter than expected by the PZMAP file.
Ignore Extra Columns	Select this check box to allow lines longer than expected by the PZMAP file and ignore the extra characters.
When using as a middle or end component in a Route:	
Use Exist cFlatPack	Click **[...]** and select the **cFlatPack** component to be used as the file parser in the dialog box.

Usage

Usage rule	**cFlatPack** can be a start, middle, or end component in a Route.
Limitation	n/a

cFtp properties

cFtp allows data exchange over remote file systems.

cFtp provides access to remote file systems over the FTP, FTPS and SFTP protocols.

cFtp Standard properties

These properties are used to configure cFtp running in the Standard Job framework.

The Standard cFtp component belongs to the Connectivity/Internet family.

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. To enter the password, click the **[...]** button next to the password field, and then in the pop-up dialog box enter the password between double quotes and click **OK** to save the settings.
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

Usage rule	**cFtp** can be a start, middle or end component in a Route.

Related scenarios

No scenario is available for the Standard version of this component yet.

cHttp properties

cHttp allows you to consume external HTTP resources as a client, and to produce Web services as a server.

cHttp provides HTTP-based endpoints for consuming and producing HTTP requests.

cHttp Standard properties

These properties are used to configure cHttp running in the Standard Job framework.

The Standard cHttp component belongs to the Connectivity/Internet family.

Basic settings

Uri	The URI of the Http resource to call.
Client	Select this option to use **cHttp** as a client to call external servers.
Server	Select this option to use **cHttp** as a server to produce Web services.
Client Configuration / Method	Select an Http request method from **GET**, **POST**, **PUT**, **DELETE**, **HEAD**, **OPTIONS**, and **TRACE** in the list.
	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**: select the **Content-Type** from *text/plain*, *text/html*, *text/xml*, *application/x-www-form-urlencoded*, *application/xml*, *application/JSON*, or *other...* (specify the Content-Type in the next field that appears when *other...* is selected),

	and 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. Select the **Content-Type** from *text/plain*, *text/html*, *text/xml*, *application/x-www-form-urlencoded*, *application/xml*, *application/JSON*, or *other...* (specify the Content-Type in the next field that appears when *other...* is selected),
	PUT: Request that the enclosed entity be stored under the supplied request URI. **Plain Text**: select the **Content-Type** from *text/plain*, *text/html*, *text/xml*, *application/x-www-form-urlencoded*, *application/xml*, *application/JSON*, or *other...* (specify the Content-Type in the next field that appears when *other...* is selected), and 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. Select the **Content-Type** from *text/plain*, *text/html*, *text/xml*, *application/x-www-form-urlencoded*, *application/xml*, *application/JSON*, or *other...* (specify the Content-Type in the next field that appears when *other...* is selected),

	DELETE: Request that the origin server delete the resource 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.
	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.
Server Configuration / Disable Stream Cache	DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body. When this check box is selected, DefaultHttpBinding will set the request input stream directly into the message body.
Server Configuration / Session Support	Select this check box to enable the session manager on the server side of Jetty.
Server Configuration / Use Transfer-Encoding	This option is enabled by default. If this check box is cleared, Jetty servlet will disable the HTTP streaming and set the content-length header on the response.

Server Configuration / Enable JMX	Select this option to enable Jetty JMX support for this endpoint. For more information about this option, see the sitehttp://camel.apache.org/jetty.html#Jetty-JettyJMXsupport.
Server Configuration / Match on URI Prefix	Select this check box to use the CamelServlet to find a target consumer by matching the URI prefix if no exact match is found. For more information about this option, see the site http://camel.apache.org/how-do-i-let-jetty-match-wildcards.html.
Server Configuration / Use Jetty Continuation	Select this check box to use Jetty continuations for the Jetty Server. For more information about this option, see the site http://wiki.eclipse.org/Jetty/Feature/Continuations.
Server Configuration / Enable HTTP TRACE	Select this option to enable the HTTP TRACE method.

Advanced settings

Headers	Click the **[+]** button to add lines as needed and define the key and value for headers.

Usage

Usage rule	**cHttp** provides Http based endpoints for consuming external Http resources, that is to say as a client to call external servers using Http.
Limitation	Due to license incompatibility, one or more JARs required to use this component are not provided. You can install the missing JARs for this particular component by clicking the **Install** button on the **Component** tab view. You can also find out and add all missing JARs easily on the **Modules** tab in the **Integration** perspective of your studio. You can find more details about how

	to install external modules in Talend Help Center (https://help.talend.com).

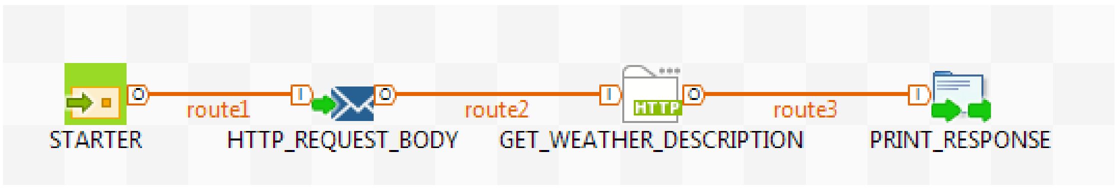
Scenario 1: Retrieving the content of a remote file

This scenario applies only to a Talend solution with ESB.

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 .

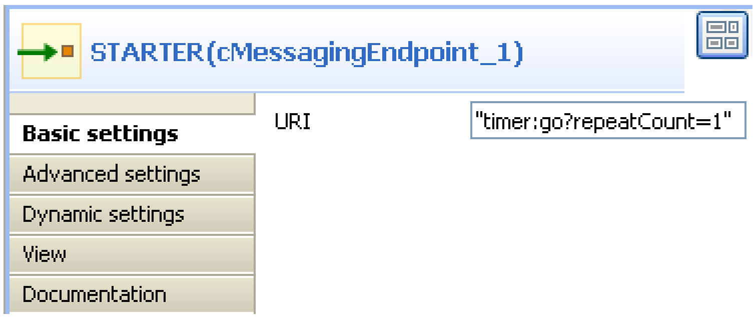
Dropping and linking the components

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.



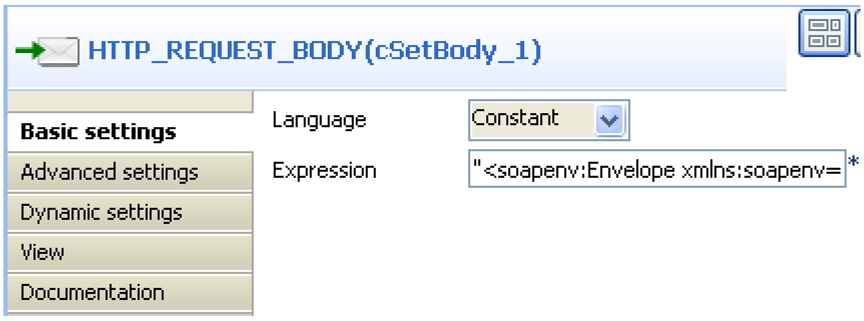
Configuring the components

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



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.



Language Constant

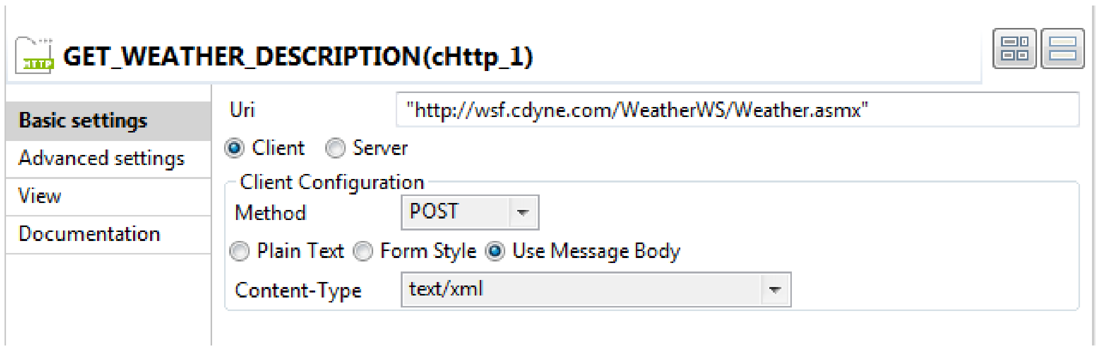
Expression "<soapenv:Envelope xmlns:soapenv=

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

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

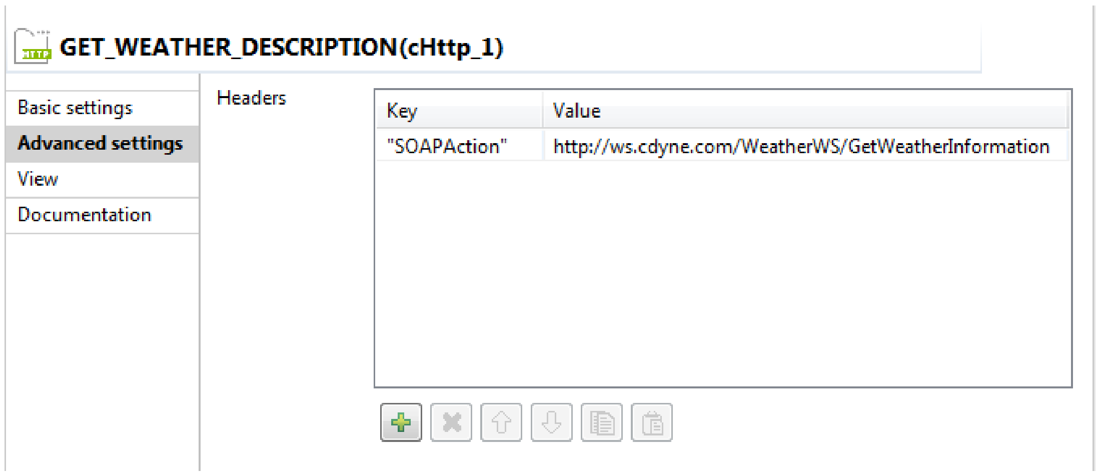
   <soapenv:Envelope xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\"
   xmlns:weat=\"http://ws.cdyne.com
   /WeatherWS/\"><soapenv:Header/
   ><soapenv:Body><weat:GetWeatherDefinitionInformation/></soapenv:Body></
   soapenv:Envelope>

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


GET_WEATHER_DESCRIPTION(cHttp_1)

Uri "http://wsf.cdyne.com/WeatherWS/Weather.asmx"
[✓ Client] Server
Client Configuration
Method POST
Plain Text Form Style [✓ Use Message Body]
Content-Type text/xml

7. In the **Uri** field, enter the location of the file to fetch, *http://wsf.cdyne.com/WeatherWS/Weather.asmx* in this example.
8. Click the **Client** radio button to use this **cHttp** component as a client.
9. Select *POST* in the **Method** list and then the **Use Message Body** radio button. Select *text/xml* in the **Content-Type** list.
10. Click **Advanced settings** for further setup.

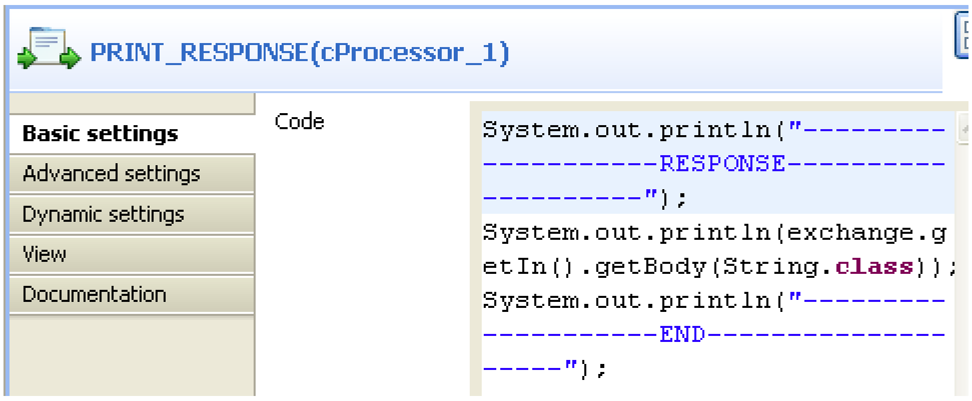

GET_WEATHER_DESCRIPTION(cHttp_1)

Headers

Key	Value
"SOAPAction"	http://ws.cdyne.com/WeatherWS/GetWeatherInformation

11. Click the **[+]** button to add a line in the **Headers** table.

    Type in `SOAPAction` and `http://ws.cdyne.com/WeatherWS/GetWeatherInformation` for the **Key** and **Value** fields.
12. 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---------------
-----");
```

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

System.out.println("--------------------RESPONSE--------------------");

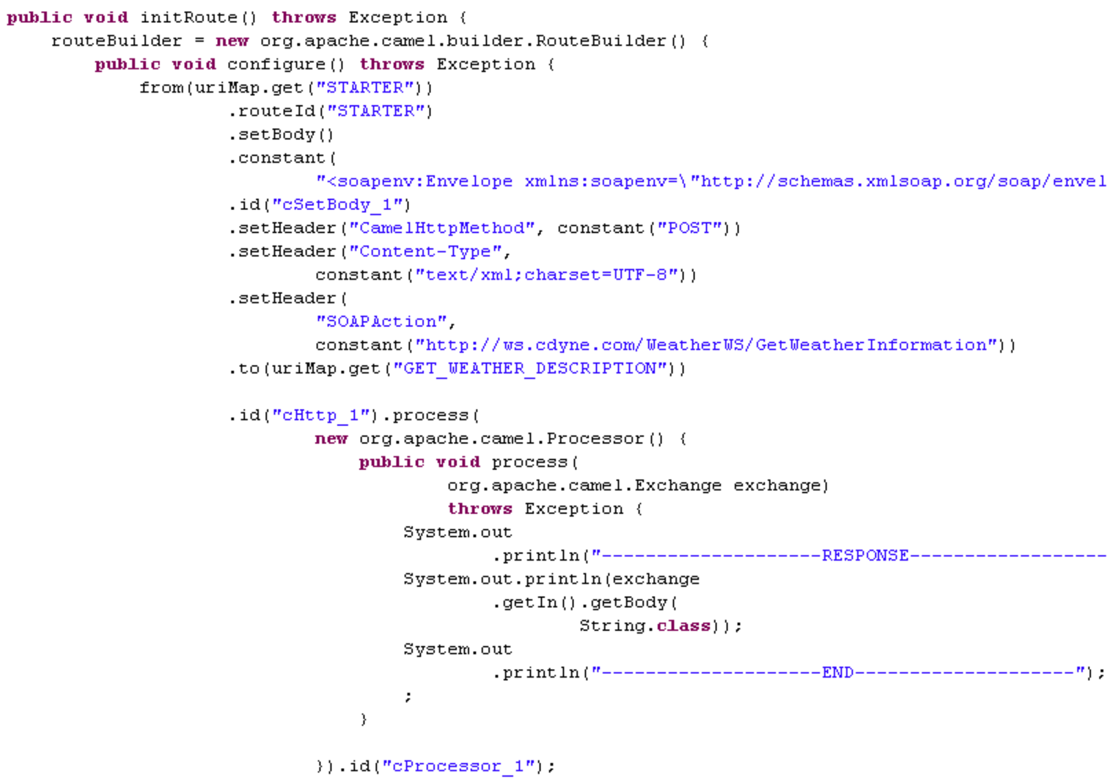
System.out.println(exchange.getIn().getBody(String.class));

System.out.println("--------------------END--------------------");

14. 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.

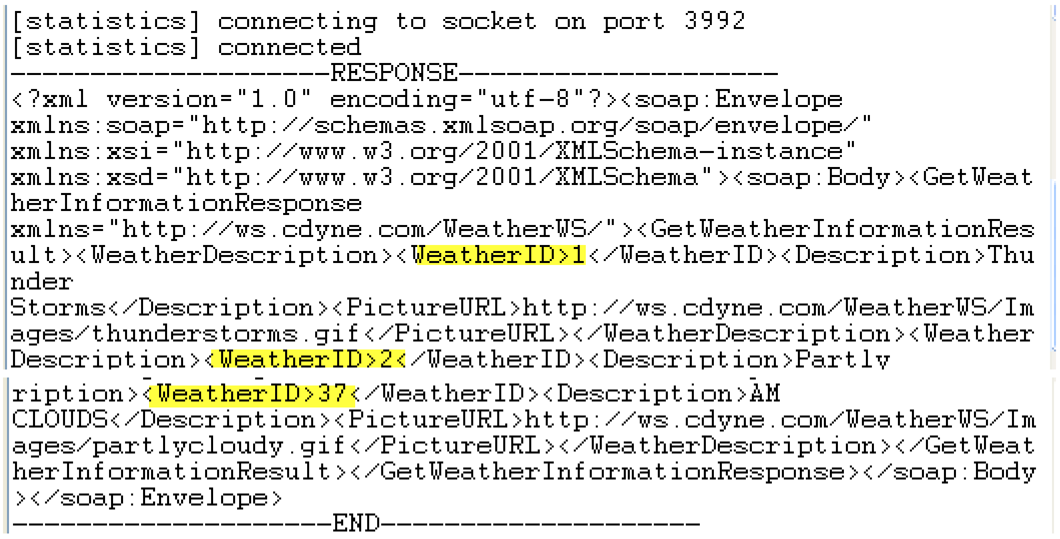


```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("STARTER"))
                    .routeId("STARTER")
                    .setBody()
                    .constant(
                            "<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.

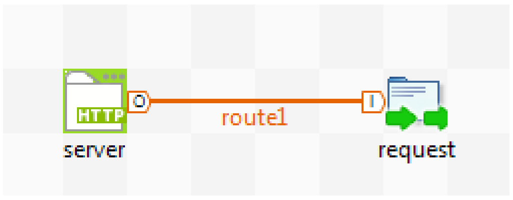
Scenario 2: Using cHttp to produce a Web service

This scenario applies only to a Talend solution with ESB.

In this scenario, **cHttp** is used to as a server to start a Web service. A **cProcessor** component is used to request the service.

Dropping and linking the components

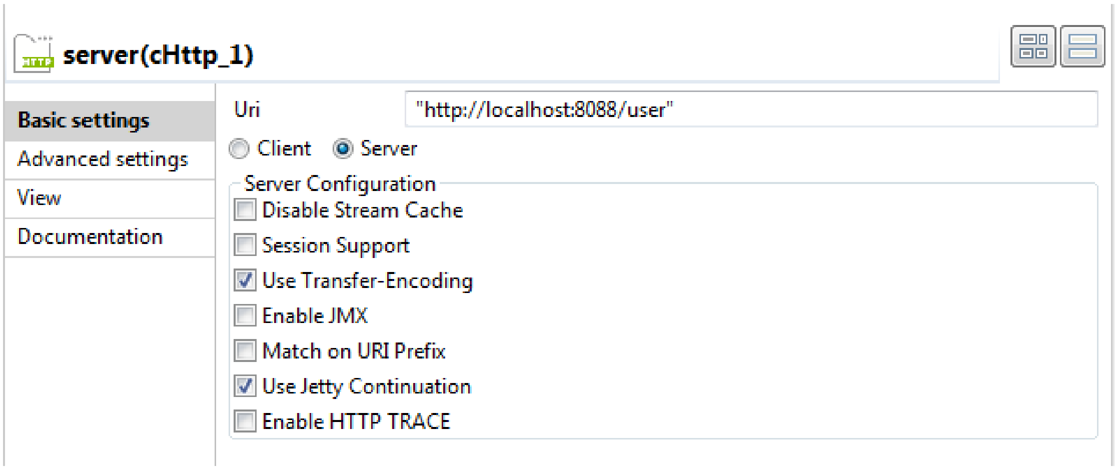
1. Drag and drop a **cHttp** and a **cProcessor** from the **Palette** onto the design workspace.



2. Link the components using a **Row** > **Route** connection.
3. Label the components for better identification of their roles.

Configuring the components

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

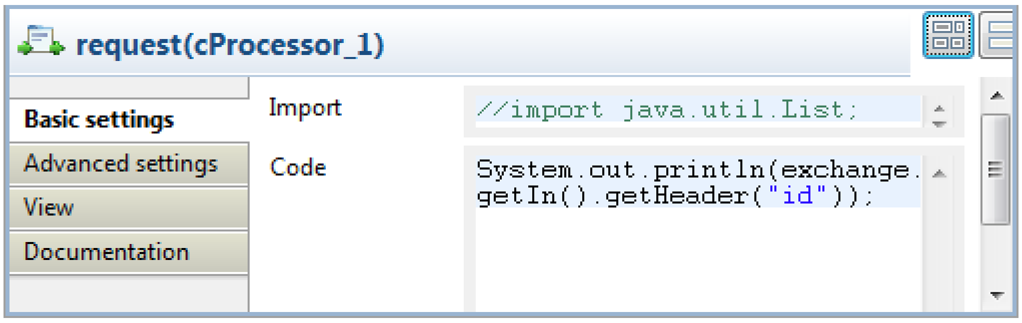


Uri "http://localhost:8088/user"

Client [✓ Server]

Server Configuration
- [ ] Disable Stream Cache
- [ ] Session Support
- [x] Use Transfer-Encoding
- [ ] Enable JMX
- [ ] Match on URI Prefix
- [x] Use Jetty Continuation
- [ ] Enable HTTP TRACE

2. In the **URI** field, enter `"http://localhost:8088/user"` where the service will be accessible for requests.
3. Click the **Server** radio button to use this **cHttp** component as a server. Keep the default settings of the other options.
4. Double-click **cProcessor** to open its **Basic settings** view in the **Component** tab.



Import
```
//import java.util.List;
```
Code
```
System.out.println(exchange.
getIn().getHeader("id"));
```

5. In the **Code** area, enter the following code to get the header `id` of the exchange message:

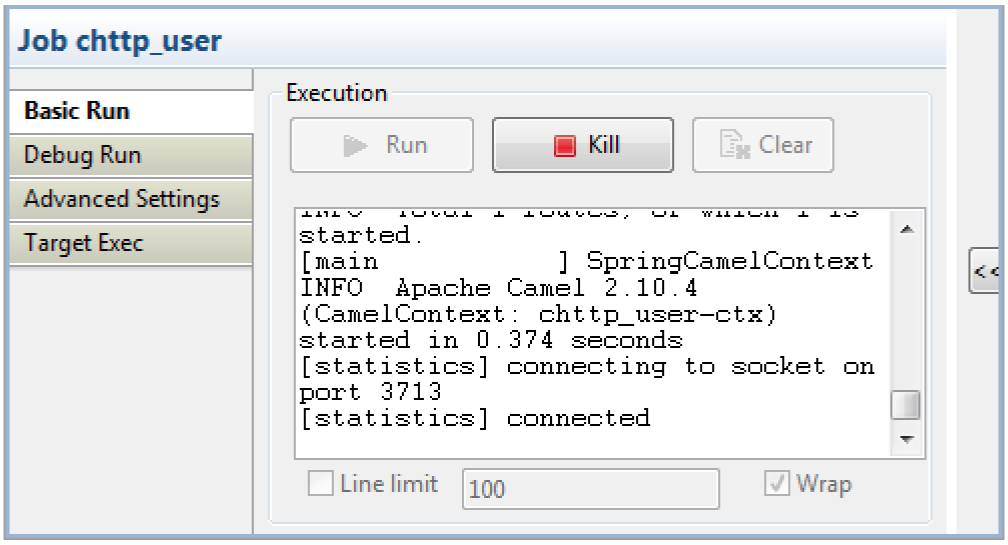
   System.out.println(exchange.getIn().getHeader("id"));
6. Press **Ctrl+S** to save your Route.

Viewing code and executing the Route

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

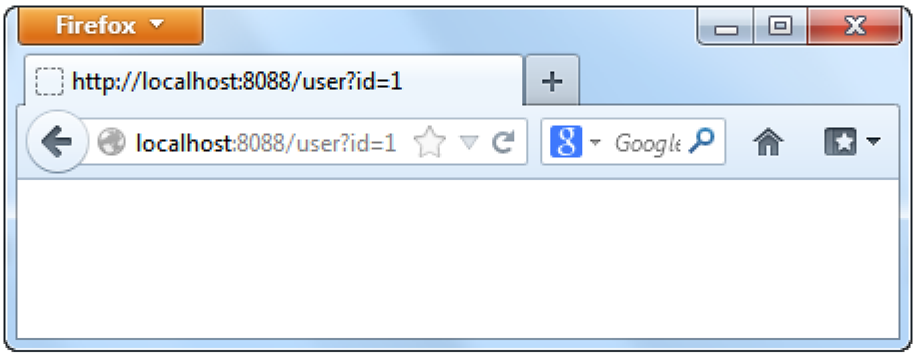
```
public void configure() throws java.lang.Exception {
     from(uriMap.get("server_cHttp_1"))

     .routeId("server_cHttp_1").process(new org.apache.camel.Processor() {
         public void process(org.apache.camel.Exchange exchange)
                 throws Exception {
             System.out.println(exchange.getIn().getHeader("id"));
         }

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

   As shown above, the route is built `from` the endpoint `server_cHttp_1` and the message header `id` is printed out by `cProcessor_1`.
2. Press **F6** to execute the Route.

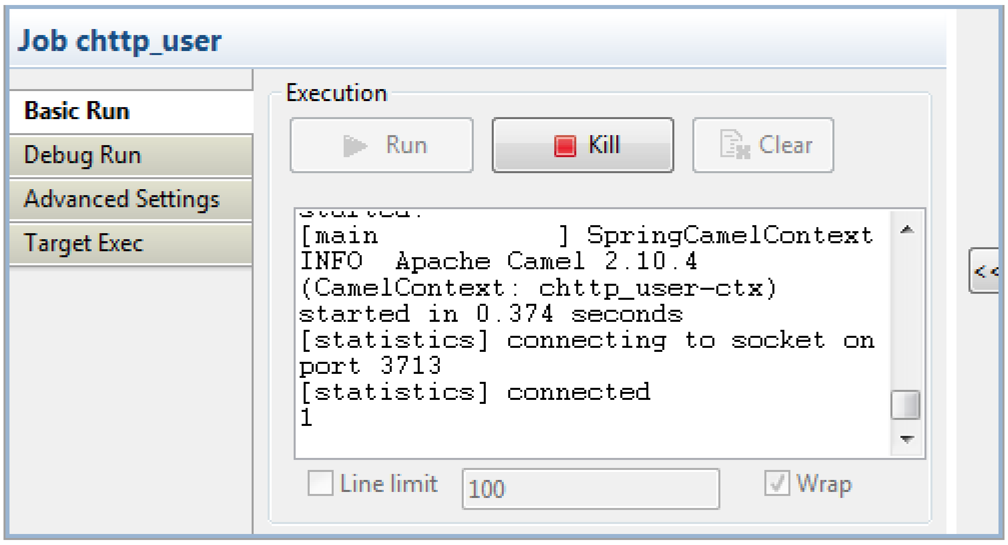


```
started.
[main         ] SpringCamelContext
INFO  Apache Camel 2.10.4
(CamelContext: chttp_user-ctx)
started in 0.374 seconds
[statistics] connecting to socket on
port 3713
[statistics] connected
```

3. Start a Web browser. In the address bar, type in `http://localhost:8088/user?id=1` and click **Enter**. A blank page opens up.



4. Switch back to the studio. The id `1` is printed in the console.



```
[main         ] SpringCamelContext
INFO  Apache Camel 2.10.4
(CamelContext: chttp_user-ctx)
started in 0.374 seconds
[statistics] connecting to socket on
port 3713
[statistics] connected
1
```

cIdempotentConsumer properties

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.

cIdempotentConsumer deduplicates messages and thereby prevents the receiving message endpoint from receiving duplicate messages.

cIdempotentConsumer Standard properties

These properties are used to configure cIdempotentConsumer running in the Standard Job framework.

The Standard cIdempotentConsumer component belongs to the Routing family.

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. **Warning:** *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. In the **Language** list, select from **None**, **Bean**, **Constant**, **CorrelationID**, **EL**, **Groovy**, **Header**, **JavaScript**, **JoSQL**, **JSonPath**, **JXPath**, **MVEL**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**.

Predicate	Type in the expression to use to specify the identifier of the messages.
Add Namespaces	This option appears when **XPath** is selected in the **Language** list. Select this check box to add namespaces for the Xpath expression. Click **[+]** to add as many namespaces as required to the table and define the prefix and URI in the corresponding columns.
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

Usage rule	**cIdempotentConsumer** is used as a middle component in a Route.
Connections	**idemp**
	Route
Limitation	n/a

cIntercept properties

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.

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.

cIntercept Standard properties

These properties are used to configure cIntercept running in the Standard Job framework.

The Standard cIntercept component belongs to the Exception Handling family.

Usage

Usage rule	**cIntercept** is a start component of a sub-route.
Connections	**Row / Route**
	Trigger / When
Limitation	To keep the original sub-routes untouched, cIntercept only be used in a separate sub-route .

cJavaDSLProcessor properties

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.

cJavaDSLProcessor implements producers and consumers of message exchanges or implements a Message Translator using the Java Domain Specific Language (DSL).

cJavaDSLProcessor Standard properties

These properties are used to configure cJavaDSLProcessor running in the Standard Job framework.

The Standard cJavaDSLProcessor component belongs to the Core family.

Basic settings

Code	Type in the code you want to implement using Java DSL.

Usage

Usage rule	**cJavaDSLProcessor** is used as a middle or end component in a Route.
Limitation	n/a

Related scenario:

For a related scenario, see Scenario: Wiretapping a message in a Route on page 214.

cJMS properties

cJMS is used to exchange messages between a Route and a JMS provider.

cJMS sends messages to, or consumes messages from, a JMS Queue or Topic.

cJMS Standard properties

These properties are used to configure cJMS running in the Standard Job framework.

The Standard cJMS component belongs to the Connectivity/Messaging family.

Basic settings

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 **cMQConnectionFactory** 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

Usage rule	**cJMS** can be a start, middle or end component in a Route. It has to be used with the **cMQConnectionFactory** component, which creates a connection to a MQ server. For more information about **cMQConnectionFactory**, see cMQConnectionFactory properties on page 146.
Limitation	n/a

cKafka properties

cKafka allows you to communicate with Apache Kafka message broker.

cKafka sends messages to, or consumes messages from Apache Kafka message broker.

cKafka Standard properties

These properties are used to configure cKafka running in the Standard Job framework.

The Standard cKafka component belongs to the Connectivity/Messaging family.

Basic settings

Broker List	Specify the list of Kafka message brokers in the form `hostname1:port1,hostname2:port2,hostname3:p`
Specify an Id string of the client to pass to the server when making requests.	
Type in a name for the message topic in the message broker.	
Enter the Id of the Kafka Connect cluster group.	

Advanced settings

Kafka Properties	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 sitehttp://kafka.apache.org/documentation.html for available options.
Click **[...]** and enter the password of the private key in the key store file in double quotes.	
Enter the location of the key store file.	
Click **[...]** and enter the password for the key store file in double quotes. This is only needed if **SSL Keystore Location** is configured.	

Enter the location of the trust store file.	
Click **[...]** and enter the password for the trust store file in double quotes.	
Enter the list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default all the available cipher suites are supported.	
Enter the endpoint identification algorithm to validate server hostname using server certificate.	
Enter the Kerberos principal name that Kafka runs as.	
Select the protocol to use to communicate with brokers from **Plaintext**, **SSL**, **SASL over Plaintext**, and **SASL over SSL**.	

Usage

Usage rule	**cKafka** is used as a start, middle or end component in a Route.
The following options are available only when the **cKafka** is used as a Producer:	
Partitioner	Enter the partitioner that determines how data is distributed across the Kafka cluster.
Serializer Class	Enter the class name of the serializer to be used.
Key Serializer Class	Enter the class name of the key serializer to be used.

Send Buffer (bytes)	The size of the TCP send buffer to use when sending data.
Request Required Acks	Specify whether the producer waits for an acknowledgement from the broker that the message was received by entering: • **0**, which means that the producer never waits for an acknowledgement from the broker; • **1**, which means that the producer gets an acknowledgement after the leader replica has received the data; • **-1** or **all**, which means that the producer gets an acknowledgement after all in-sync replicas have received the data.
Request Timeout (ms)	Specify the maximum amount of time in milliseconds that the client will wait for the response of a request. If the response is not received before the timeout elapses, the client will resend the request if necessary or fail the request if retries are exhausted.
Compression Codec	Select compression type from **NONE**, **GZIP**, **SNAPPY**, and **LZ4**.
Buffer Memory Size	Specify the total bytes of memory the producer can use to buffer records waiting to be sent to the server. If records are sent faster than they can be delivered to the server, the producer will block for **Max Block (ms)** after which it will throw an exception.
Retries	Specify a value greater than `0` for the client to resend any record that failed to be sent with a potentially transient error.
Retry Backoff (ms)	Specify the amount of time to wait before attempting to retry a failed request to a given

	topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.
Batch Size	The producer will attempt to batch records together into fewer requests whenever multiple records are being sent to the same partition. This helps performance on both the client and the server. Specify the default batch size in bytes in this field.
Connection Idle Max (ms)	Specify the time in milliseconds after which idle connections will be closed.
Linger (ms)	Specify how long, in milliseconds, the producer must wait to group together any records that arrive into a single batched request, in order to reduce the number of requests. Note that, if the producer has received the amount of records specified by the Bath Size, it will send out the records immediately, regardless of what is specified in this setting.
Max Block (ms)	Specify the maximum amount of time in milliseconds that the producer will wait either because the buffer is full or metadata unavailable.
Max Request Size	Specify the maximum size of a request in bytes. This setting will limit the number of record batches the producer will send in a single request to avoid sending huge requests.
Receive Buffer (bytes)	Specify the size of the TCP receive buffer to use when reading data. If the value is `-1`, the OS default will be used.
Max in Flight Request	Specify the maximum number of unacknowledged requests that the client will send on a single connection before blocking. Note that

	if this setting is set to be greater than 1 and there are failed sends, there is a risk of message re-ordering due to retries if retries are enabled.
Metadata Max Age (ms)	Specify the period of time in milliseconds after which a refresh of metadata occurs even if there are no partition leadership changes to proactively discover any new brokers or partitions.
Reconnect Backoff (ms)	Specify the amount of time in milliseconds to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop.
The following options are available only when the **cKafka** is used as a Consumer:	
Send Buffer (bytes)	Specify the size of the TCP send buffer to use when sending data. If the value is `-1`, the OS default will be used.
Retry Backoff (ms)	Specify the amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.
Connection Idle Max (ms)	Specify the time in milliseconds after which idle connections will be closed.
Receive Buffer (bytes)	Specify the size of the TCP receive buffer to use when reading data. If the value is `-1`, the OS default will be used.
Metadata Max Age (ms)	Specify the period of time in milliseconds after which a refresh of metadata occurs even if there are no partition leadership changes to proactively discover any new brokers or partitions.

Reconnect Backoff (ms)	Specify the amount of time in milliseconds to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop.
Barrier Await Timeout (ms)	Specify the amount of time in milliseconds that the batching consumer task waits if the message exchange exceeds the batch size. The default is `10000`.
Auto Commit Enable	Select this check box to periodically commit the offset of messages in the background. Specify the frequency in milliseconds in the **Auto Commit Interval (ms)** that the consumer offsets are committed to Kafka.
Fetch Min (bytes)	Specify the minimum amount of data in bytes that the server should return for a fetch request. If insufficient data is available, the request will wait for that much data to accumulate before answering the request. The default setting of `1` byte means that fetch requests are answered as soon as a single byte of data is available or the fetch request times out waiting for data to arrive. Setting this to something greater than `1` will cause the server to wait for larger amounts of data to accumulate which can improve server throughput a bit at the cost of some additional latency.
Fetch Wait Max (ms)	Specify the maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by **Fetch Min (bytes)**.
Auto Offset Reset	Choose what to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server from the following:

	• **EARLIEST**: automatically reset the offset to the earliest offset; • **LATEST**: automatically reset the offset to the latest offset; • **NONE**: throw exception to the consumer if no previous offset is found for the consumer's group.
Heartbeat Interval (ms)	Specify the expected time in milliseconds between heartbeats to the consumer coordinator when using Kafka's group management facility. Heartbeats are used to ensure that the consumer's session stays active and to facilitate rebalancing when new consumers join or leave the group. This value must be set lower than **Session Timeout (ms)**, but typically should be set no higher than *1/3* of that value. It can be adjusted even lower to control the expected time for normal rebalances.
Maximum Partition Fetch (bytes)	Specify the maximum amount of data per-partition in bytes that the server will return. If the first message in the first non-empty partition of the fetch is larger than this limit, the message will still be returned to ensure that the consumer can make progress.
Session Timeout (ms)	Specify the timeout in milliseconds used to detect consumer failures when using Kafka's group management facility. The consumer sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, the broker will remove this consumer from the group and initiate a rebalance.

Partition Assignor	Specify the class name of the partition assignment strategy that the client will use to distribute partition ownership amongst consumer instances when group management is used.
Request Timeout (ms)	Specify the maximum amount of time in milliseconds that the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.

Related scenarios

No scenario is available for the Standard version of this component yet.

cLoadBalancer properties

cLoadBalancer allows you to distribute messages among several endpoints using a variety of load balancing strategies.

cLoadBalancer distributes the messages it received to multiple endpoints according to the load balancing policy.

cLoadBalancer Standard properties

These properties are used to configure cLoadBalancer running in the Standard Job framework.

The Standard cLoadBalancer component belongs to the Routing family.

Basic settings

Strategy	Select between **Random**, **Round Robin**, **Sticky**, **Topic**, **Failover**, and **Custom**. Each method is described below.
The receiving endpoint is chosen randomly at each exchange.	
Messages are distributed according to the round robin method which distributes the load evenly.	
Language	Select the language of the expression to use in the **Expression** field to distribute the messages from **None**, **Bean**, **Constant**, **CorrelationID**,

	EL, **Groovy**, **Header**, **JavaScript**, **JoSQL**, **JSonPath**, **JXPath**, **MVEL**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**.
Expression	Type in the expression that will be used to calculate a correlation key that will determine the endpoint to choose.
Select this option to send all the messages to all the endpoints.	
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.
Load balancer	Type in the name of your custom load balancer.

Usage

Usage rule	**cLoadBalancer** is used as a middle component in a Route.
Connections	**Load Balance**
	Route
Limitation	n/a

cLog properties

cLog is used to log message exchanges.

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.

cLog Standard properties

These properties are used to configure cLog running in the Standard Job framework.

The Standard cLog component belongs to the Miscellaneous family.

Basic settings

Logging Category	Enter the name of the logging category to use.
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	Select this option to use throughput logging and specify a group size for the throughput logging.

throughput logging (For default output log message only)	**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 specify the output log message. **Message**: Use Simple language to construct a dynamic message which gets logged.
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

Usage rule	**cLog** is used as a middle or end component in a Route.
Limitation	n/a

Related scenario:

For a related scenario, see Scenario: Routing messages according to a criterion on page 118.

cLoop properties

cLoop is used to process a message or messages repetitively.

cLoop allows you to process a message or messages a number of times and possibly in different ways.

cLoop Standard properties

These properties are used to configure cLoop running in the Standard Job framework.

The Standard cLoop component belongs to the Orchestration family.

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. The **Add Namespaces** option appears when **XPath** is selected in the **Language** list. Select this check box to add namespaces for the Xpath expression. Click **[+]** to add as many namespaces as required to the table and define the prefix and URI in the corresponding columns.
	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.
Copy	Select this check box to use the copy mode. It is cleared by default. If this option is disabled, the same exchange will be used for each iteration. So the result from the previous iteration will be visible for the next. If this option is enabled, each iteration restarts with a fresh copy of the input exchange.

Usage

Usage rule	**cLoop** can be a middle component in a Route.
Limitation	n/a

Related scenarios

No scenario is available for the Standard version of this component yet.

cMail properties

cMail is designed to send or receive mails in a Route.

cMail provides access to Email via Spring's Mail support and the underlying Java Mail system.

cMail Standard properties

These properties are used to configure cMail running in the Standard Job framework.

The Standard cMail component belongs to the Connectivity/Internet family.

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. To enter the password, click the **[...]** button next to the password field, and then in the pop-up dialog box enter the password between double quotes and click **OK** to save the settings.
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

Usage rule	When used as a start component, **cMail** is intended to receive mails. Otherwise, it is intended to send mails.
Limitation	Due to license incompatibility, one or more JARs required to use this component are not provided. You can install the missing JARs for this particular component by clicking the **Install** button on the **Component** tab view. You can also find out and add all missing JARs easily on the **Modules** tab in the **Integration** perspective of your studio. You can find more details about how to install external modules in Talend Help Center (https://help.talend.com).

Scenario: Using **cMail** to send and receive mails

This scenario applies only to a Talend solution with ESB.

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

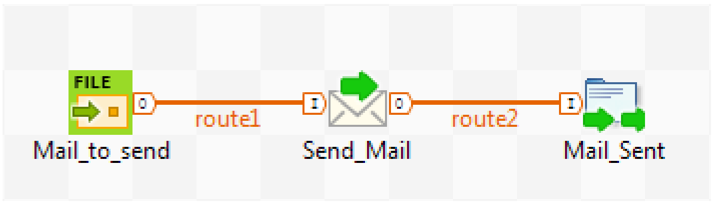
Now we build a route to send a mail.

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

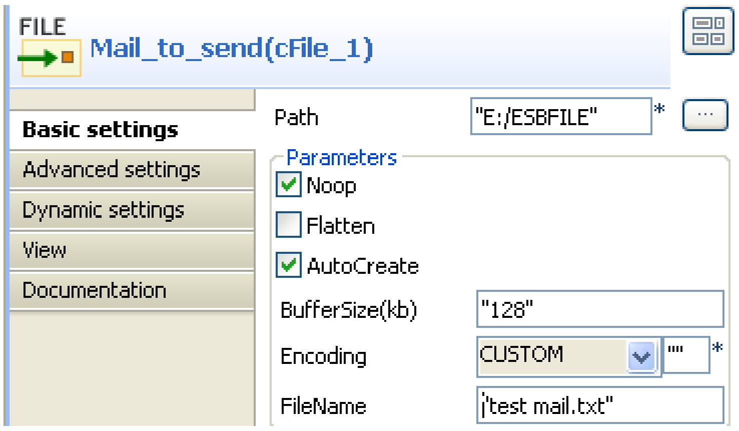
Now we build a route to receive the mail.

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.



Path "E:/ESBFILE"

Parameters

- [✓] Noop
- [ ] Flatten
- [✓] AutoCreate

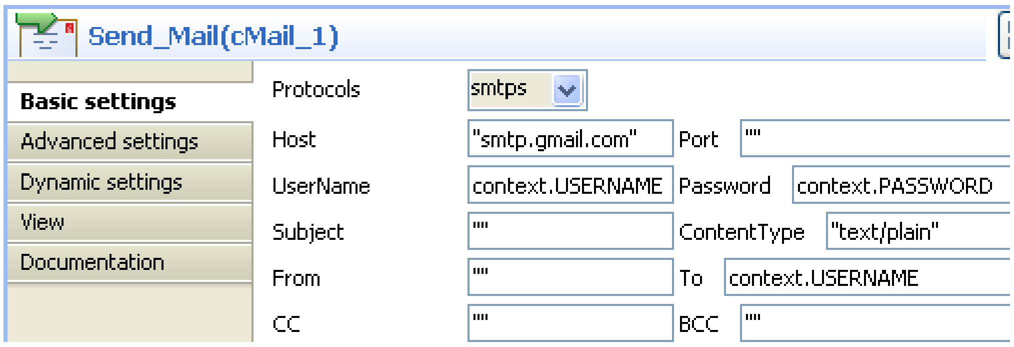
BufferSize(kb) "128"

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.



Protocols	smtps		
Host	"smtp.gmail.com"	Port	""
UserName	context.USERNAME	Password	context.PASSWORD
Subject	""	ContentType	"text/plain"
From	""	To	context.USERNAME
CC	""	BCC	""

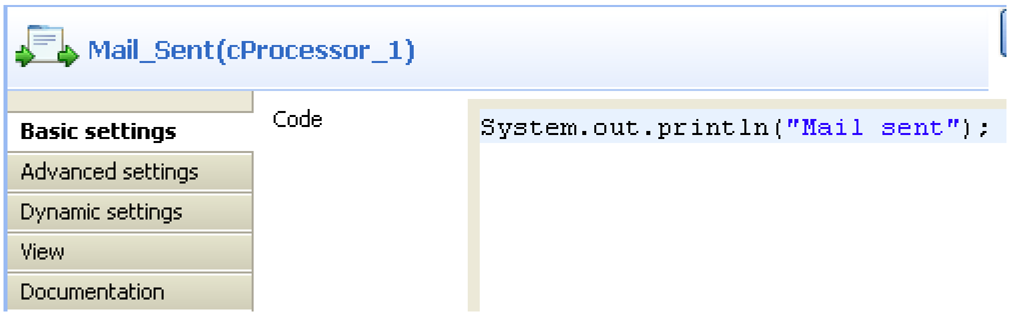
7. In the **Protocols** list, select *smtps*.

   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 Studio User Guide*.

   Keep the default setting of the **ContentType** field, *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.

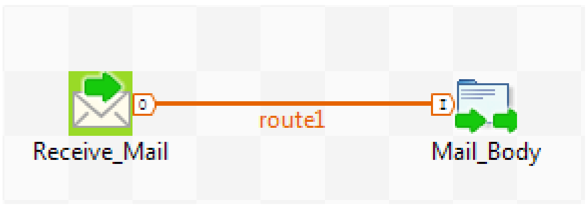
   System.out.println("Mail sent");

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

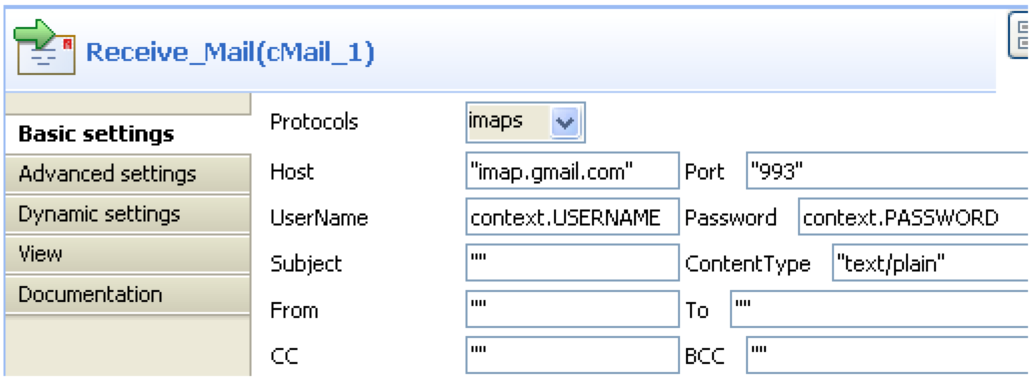
```
[statistics] connecting to socket on port 3612
[statistics] connected
Mail sent
```

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.

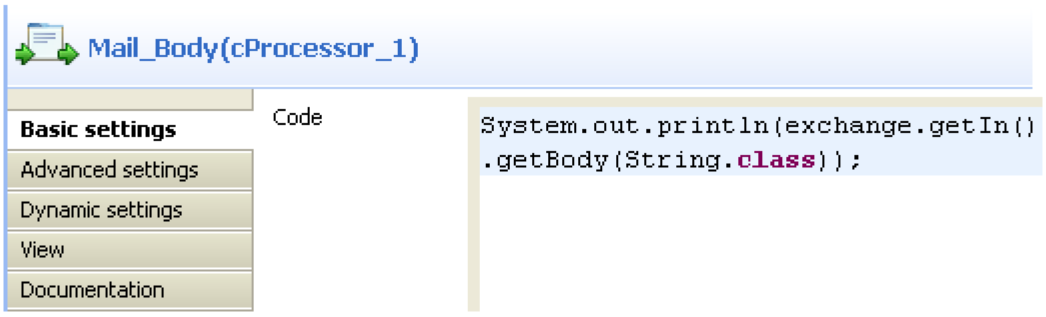


3. Double-click **cMail** to open its **Basic settings** view in the **Component** tab.



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 Studio User Guide*.
8. Keep the default setting of the **ContentType** field, *text/plain*.
9. Double-click **cProcessor** to open its **Basic settings** view in the **Component** tab.

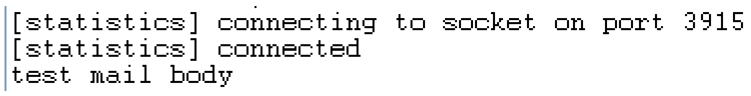


```
System.out.println(exchange.getIn()
.getBody(String.class));
```

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*.

cMessageFilter properties

Use **cMessageFilter** to eliminate unwanted messages from a channel according to the defined criterion.

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.

cMessageFilter Standard properties

These properties are used to configure cMessageFilter running in the Standard Job framework.

The Standard cMessageFilter component belongs to the Routing family.

Basic settings

Language	Select the language of the expression you use to filter your messages from **None**, **Bean**, **Constant**, **CorrelationID**, **EL**, **Groovy**, **Header**, **JavaScript**, **JoSQL**, **JSonPath**, **JXPath**, **MVEL**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**.
Expression	Type in the expression to use to filter the messages.

Add Namespaces	This option appears when **XPath** is selected in the **Language** list. Select this check box to add namespaces for the Xpath expression. Click **[+]** to add as many namespaces as required to the table and define the prefix and URI in the corresponding columns.

Usage

Usage rule	**cMessageFilter** is used as a middle component in a Route.
Connections	**Filter**
	Route
Limitation	n/a

cMessageRouter properties

cMessageRouter creates different channels for each filtered message types so that messages can later on be treated more accurately in each new channel.

cMessageRouter routes messages in different channels according to specified conditions.

cMessageRouter Standard properties

These properties are used to configure cMessageRouter running in the Standard Job framework.

The Standard cMessageRouter component belongs to the Routing family.

Usage

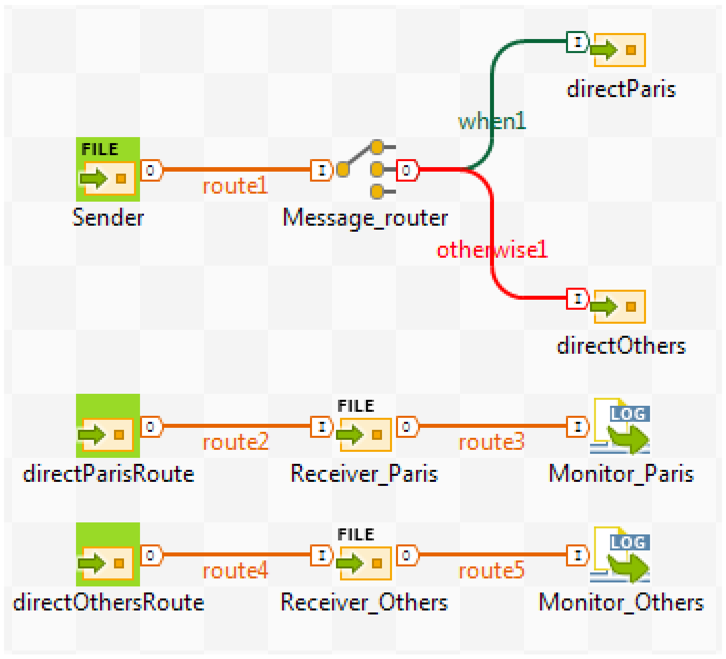
Usage rule	**cMessageRouter** is used as a middle component in a Route. It can only have one input channel but multiple output channels. Messages can be outputted through either a **When**, **Otherwise** or **Route** types of connection.
Connections	**Row / Route**
	Trigger / When
	Trigger / Otherwise

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

This scenario applies only to a Talend solution with ESB.

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

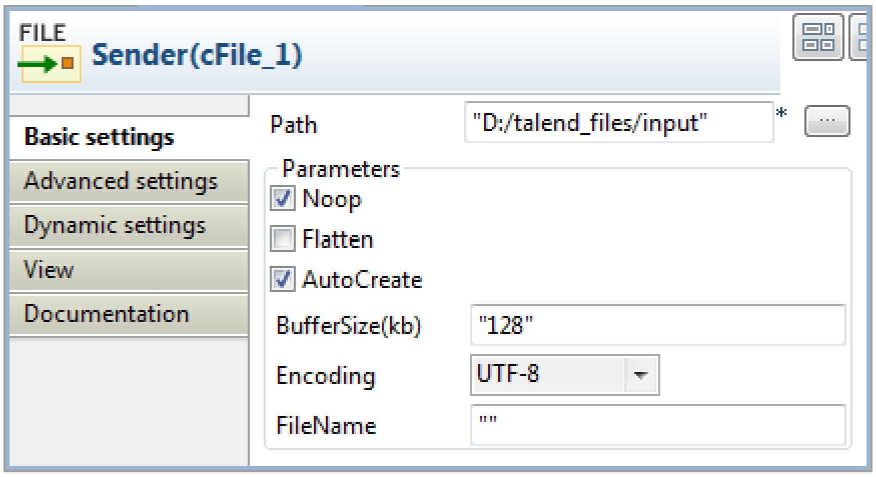
1. From the **Connectivity** folder of the **Palette**, drop three **cFile** and four **cMessagingEndpoint** components onto the design workspace, and label them *Sender*, *Receiver_Paris*, and *Receiver_Others*, *directParis*, *directOthers*, *directParisRoute*, and *directOthersRoute* respectively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

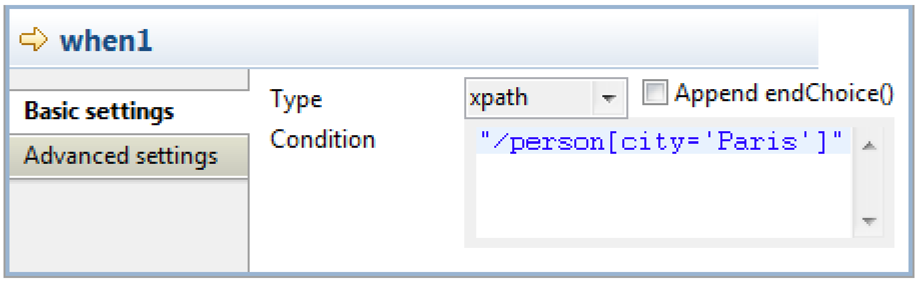

FILE
Sender(cFile_1)

Basic settings | Advanced settings | Dynamic settings | View | Documentation

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.



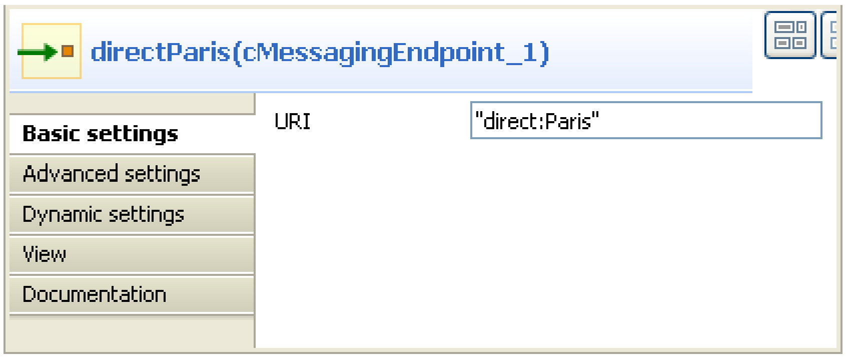
Type: xpath

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.

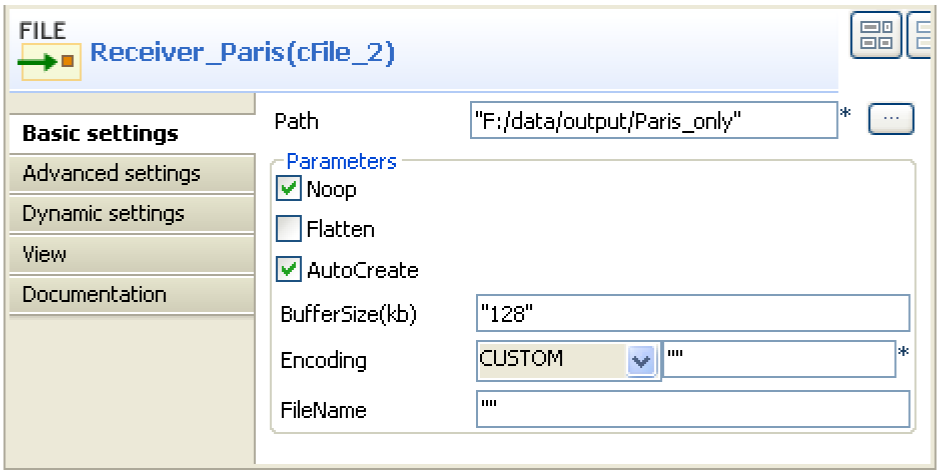


URI: "direct:Paris"

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.



Path: "F:/data/output/Paris_only"

[✓] Noop

[ ] Flatten

[✓] AutoCreate

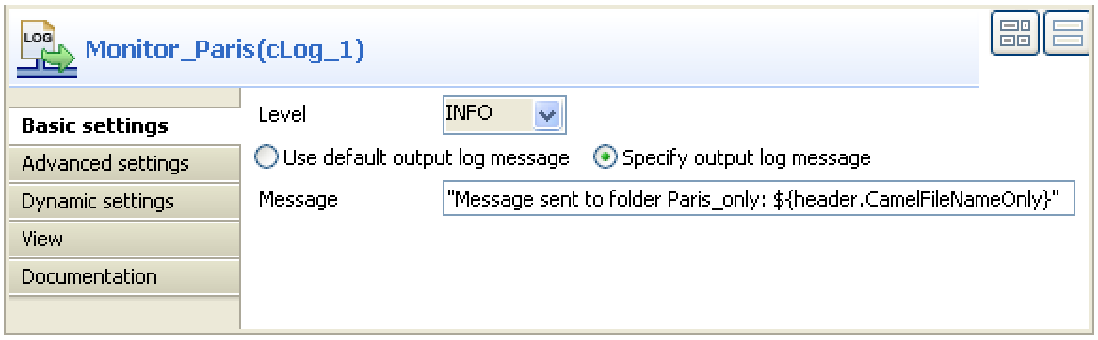
BufferSize(kb): "128"

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.



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

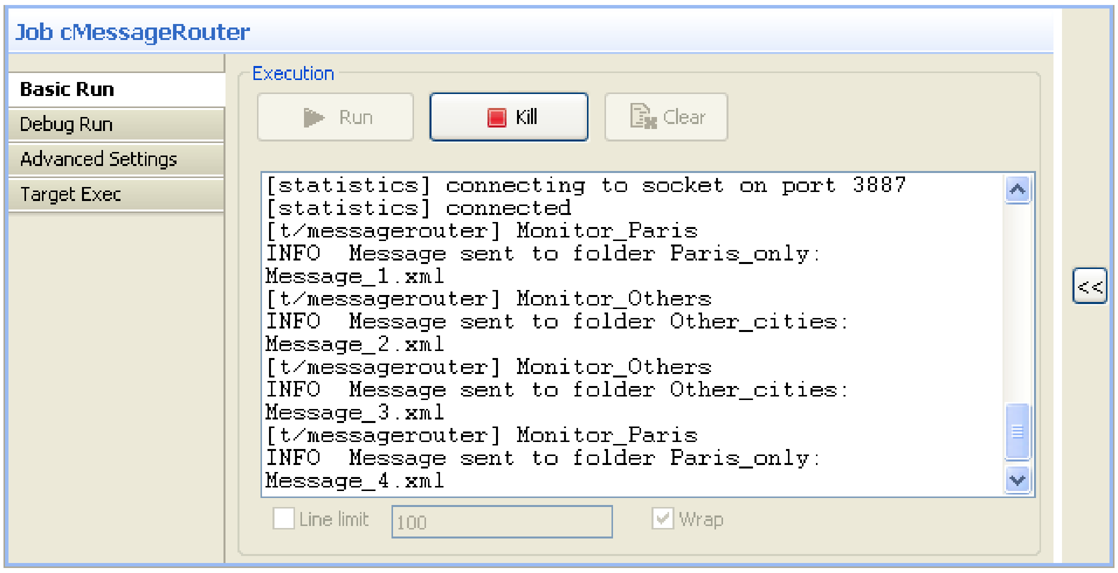
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

```
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
```

cMessagingEndpoint properties

cMessagingEndpoint sends or receives messages.

cMessagingEndpoint allows two applications to communicate by either sending or receiving messages, one endpoint can not do both.

Commonly used Camel components

The following table lists the most commonly used Camel components that can be called by cMessagingEndpoint. Click the links in the table to go to the Apache Camel's Website for the latest information of the these components. Make sure to use the information applicable for the Camel Version included in Talend ESB. See also *Talend ESB Mediation Developer Guide* for details of the Camel components.

Component / ArtifactId / URI	Description
ActiveMQ / activemq-camel `activemq:[topic:]destinationName`	For JMS Messaging with Apache ActiveMQ
AHC / camel-ahc `ahc:http[s]://hostName[:port][/resourceUri][?options]`	To call external HTTP services using Async Http Client.
APNS / camel-apns `apns:<notify|consumer>[?options]`	For sending notifications to Apple iOS devices.
Avro / camel-avro `avro:[transport]:[host]:[port][/messageName][?options]`	Working with Apache Avro for data serialization.

Component / ArtifactId / URI	Description
Atom / camel-atom `atom:uri`	Working with Apache Abdera for atom integration, such as consuming an atom feed.
Bean / camel-core `bean:beanName[?method=someMethod]`	Uses the Camel Bean Binding to bind message exchanges to beans in the Camel Registry. Is also used for exposing and invoking POJO (Plain Old Java Objects).
Cache / camel-cache `cache://cachename[?options]`	The cache component facilitates creation of caching endpoints and processors using EHCache as the cache implementation.
Class / camel-core `class:className[?method=someMethod]`	Uses the Camel Bean Binding to bind message exchanges to beans in the Camel Registry. Is also used for exposing and invoking POJOs (Plain Old Java Objects).
CMIS / camel-cmis `cmis://cmisServerUrl[?options]`	Uses the Apache Chemistry client API to interface with CMIS supporting CMS.
Context / camel-context `context:camelContextI localEndpointName`	Used to refer to endpoints within a separate CamelContext to provide a simple black box composition approach so that routes can be combined into a CamelContext and then used as a black box component inside other routes in other CamelContexts
CouchDB / camel-couchdb `couchdb:hostName[:port]/database[?options]`	To integrate with Apache CouchDB.
Crypto (Digital Signatures) / camel-crypto `crypto:sign:name[?options], crypto:verify:name[?options]`	Used to sign and verify exchanges using the Signature Service of the Java Cryptographic Extension.
CXF / camel-cxf `cxf:address[?serviceClass=...]`	Working with Apache CXF for web services integration
CXF Bean / camel-cxf `cxf:bean name`	Process the exchange using a JAX WS or JAX RS annotated bean from the registry. Requires less configuration than the above CXF Component
CXFRS / camel-cxf `cxfrs:address[?resourcesClasses=...]`	Working with Apache CXF for REST services integration
Direct / camel-core `direct:name`	Synchronous call to another endpoint from same CamelContext

Component / ArtifactId / URI	Description
Disruptor / camel-disruptor `disruptor-vm:someName[?<option>]`	To provide the implementation of SEDA which is based on disruptor.
ElasticSearch / camel-elasticsearch `elasticsearch://clusterName[?options]`	Uses the Bean Binding to bind message exchanges to EJBs. It works like the Bean component but just for accessing EJBs. Supports EJB 3.0 onwards.
Spring Event / camel-spring `event://default, spring-event://default`	Working with Spring ApplicationEvents
Exec / camel-exec `exec://executable[?options]`	For executing system commands
Facebook / camel-facebook `facebook://endpoint[?options]`	Providing access to all of the Facebook APIs accessible using Facebook4J
File / camel-core `file://nameOfFileOrDirectory`	Sending messages to a file or polling a file or directory.
Flatpack / camel-flatpack `flatpack:[fixed|delim]:configFile`	Processing fixed width or delimited files or messages using the FlatPack library
FOP / camel-fop `fop:outputFormat[?options]`	Renders the message into different output formats using Apache FOP.
FreeMarker / camel-freemarker `freemarker:someTemplateResource`	Generates a response using a Freemarker template
FTP / camel-ftp `ftp:contextPath[?options]`	Sending and receiving files over FTP.
FTPS / camel-ftp `ftps://[username@]hostName[:port]/directoryName[?options]`	Sending and receiving files over FTP Secure (TLS and SSL).
Geocoder / camel-geocoder `geocoder:address|latlng:latitude,longitude>[?options]`	Supports looking up geocoders for an address, or reverse lookup geocoders from an address.

Component / ArtifactId / URI	Description
Google Guava EventBus / camel-guava-eventbus `guava-eventbus:busName[?options]`	The Google Guava EventBus allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other). This component provides integration bridge between Camel and Google Guava EventBus infrastructure.
HBase / camel-hbase `hbase://table[?options]`	For reading/writing from/to an HBase store (Hadoop database).
HDFS / camel-hdfs `hdfs://hostName[:port][/path][?options]`	For reading/writing from/to an HDFS file system using Hadoop 1.x.
HDFS2 / camel-hdfs2 `hdfs2://hostName[:port][/path][?options]`	For reading/writing from/to an HDFS file system using Hadoop 2.x.
HL7 / camel-hl7 `mina2:tcp://hostname[:port]`	For working with the HL7 MLLP protocol and the HL7 model using the HAPI library.
HTTP4 / camel-http4 `http4://hostname[:port]`	For calling out to external HTTP servers using Apache HTTP Client 4.x
IMAP / camel-mail `imap://hostname[:port]`	Receiving email using IMap
Infinispan / camel-infinispan `infinispan://hostName[?options]`	For reading/writing from/to Infinispan distributed key/value store and data grid
Jasypt / camel-jasypt `jasypt: uri`	Simplified on-the-fly encryption library, integrated with Camel.
JCR / camel-jcr `jcr://user:password@repository/path/to/node`	Storing a message in a JCR (JSR-170) compliant repository like Apache Jackrabbit
JDBC / camel-jdbc `jdbc:dataSourceName?options`	For performing JDBC queries and operations
Jetty / camel-jetty `jetty:url`	For exposing services over HTTP
JGroups / camel-jgroups `jgroups:clusterName[?options]`	The `jgroups:` component provides exchange of messages between Camel infrastructure and JGroups clusters.

Component / ArtifactId / URI	Description
JMS / camel-jms `jms:[topic:]destinationName`	Working with JMS providers
JMX / camel-jmx `jmx://platform?options`	For working with JMX notification listeners
JPA / camel-jpa `jpa://entityName`	For using a database as a queue via the JPA specification for working with OpenJPA, Hibernate or TopLink
Jsch / camel-jsch `scp://localhost/destination`	Support for the scp protocol.
Kafka / camel-kafka `kafka://server:port[?options]`	For producing to or consuming from Apache Kafka message brokers.
Krati / camel-krati `krati://[path to datastore/][?options]`	For producing to or consuming to Krati datastores.
Log / camel-core `log:loggingCategory[?level=ERROR]`	Uses Jakarta Commons Logging to log the message exchange to some underlying logging system like log4j
Lucene / camel-lucene `lucene:searcherName:insert [?analyzer=<analyzer>]`	Uses Apache Lucene to perform Java-based indexing and full text based searches using advanced analysis/tokenization capabilities
MINA2 / camel-mina2 `mina2:[tcp|udp|vm]:host[:port][?options]`	Working with Apache MINA 2.x.
Mock / camel-core `mock:name`	For testing routes and mediation rules using mocks
MongoDB / camel-mongodb `mongodb:connectionBean[?options]`	Interacts with MongoDB databases and collections. Offers producer endpoints to perform CRUD-style operations and more against databases and collections, as well as consumer endpoints to listen on collections and dispatch objects to Camel routes.
MongoDB GridFS / camel-mongodb-gridfs `gridfs:connectionBean?database=databaseName&bucket=bucketName[&moreOptions...]`	MongoDB Gridfs component supports the producer and consumer case to interact with MongoDB via Gridfs.

Component / ArtifactId / URI	Description
MQTT / camel-mqtt `mqtt:name[?options]`	Component for communicating with MQTT M2M message brokers
Mustache / camel-mustache `mustache:templateName[?options]`	Generates a response using a Mustache template.
MyBatis / camel-mybatis `mybatis://statementName`	Performs a query, poll, insert, update or delete in a relational database using MyBatis.
Netty / camel-netty-http `netty-http:http:[port]/context-path[?options]`	Works as Netty HTTP server and client using the Netty project.
OptaPlanner / camel-optaplanner `optaplanner:solverConfig[?options]`	Solves the planning problem contained in a message with OptaPlanner.
POP3 / camel-mail `pop3://user-info@host:port`	Receiving email using POP3 and JavaMail.
Properties / camel-core `properties://key[?options]`	The properties component facilitates using property placeholders directly in endpoint uri definitions.
Quartz / camel-quartz `quartz://groupName/timerName`	Provides a scheduled delivery of messages using the Quartz scheduler
Quartz2 / camel-quartz2 `quartz2://groupName/timerName[?options]`	Provides a scheduled delivery of messages using the Quartz 2.x scheduler.
RabbitMQ / camel-rabbitmq `rabbitmq://hostname[:port]/exchangeName[?options]`	Component for integrating with RabbitMQ.
Ref / camel-core `ref:name`	Component for lookup of existing endpoints bound in the Camel Registry.
RMI / camel-rmi `rmi://host[:port]`	Working with RMI
RSS / camel-rss `rss:uri`	Working with ROME for RSS integration, such as consuming an RSS feed.
Salesforce / camel-salesforce `salesforce:topic[?options]`	To integrate with Salesforce.

Component / ArtifactId / URI	Description
SAP NetWeaver / camel-sap-netweaver `sap-netweaver:hostName[:port][?options]`	To integrate with SAP NetWeaver Gateway.
SEDA / camel-core `seda:name`	Asynchronous call to another endpoint in the same Camel Context
SERVLET / camel-servlet `servlet:uri`	For exposing services over HTTP through the servlet which is deployed into the Web container.
SJMS / camel-sjms `sjms:[queue:|topic:]destinationName[?options]`	A ground up implementation of a JMS client.
SFTP / camel-ftp `sftp://host[:port]/fileName`	Sending and receiving files over SFTP (FTP over SSH).
SMTP / camel-mail `smtp://user-info@host[:port]`	Sending email using SMTP and JavaMail
SMPP / camel-smpp `smpp://user-info@host[:port]?options`	To send and receive SMS using Short Messaging Service Center using the JSMPP library
SNMP / camel-snmp `snmp://host[:port]?options`	Polling OID values and receiving traps using SNMP via SNMP4J library
Solr / camel-solr `solr://hostName[:port]/solr[?options]`	Uses the Solrj client API to interface with an Apache Lucene Solr server.
Splunk / camel-splunk `splunk://[endpoint]?[options]`	For working with Splunk
SpringBatch / camel-spring-batch `spring-batch:jobName[?options]`	To bridge Camel and Spring Batch.
SpringIntegration / camel-spring-integration `spring-integration:defaultChannelName`	The bridge component of Camel and Spring Integration

Component / ArtifactId / URI	Description
Spring LDAP / camel-spring-ldap `spring-ldap:springLdapTemplateBean[?options]`	Camel wrapper for Spring LDAP
Spring Redis / camel-spring-redis `spring-redis://hostName:port[?options]`	Component for consuming and producing from Redis key-value store Redis
Spring Web Services / camel-spring-ws `spring-ws:[mapping-type:]address[?options]`	Client-side support for accessing web services, and server-side support for creating your own contract-first web services using Spring Web Services
SQL / camel-sql `sql:select * from table where id=#`	Performing SQL queries using JDBC
SSH / camel-ssh `ssh:[username[:password]@]host[:port][?options]`	For sending commands to a SSH server
StAX / camel-stax `stax:(contentHandlerClassName|#myHandler)`	Process messages through a SAX ContentHandler.
Stomp / camel-stomp `stomp:queue:destinationName[?options]`	For communicating with Stomp compliant message brokers, like Apache ActiveMQ or ActiveMQ Apollo
Stub / camel-core `stub:someOtherCamelUri`	Allows you to stub out some physical middleware endpoint for easier testing or debugging
Test / camel-spring `test:expectedMessagesEndpointUri`	Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint
Timer / camel-core `timer://name`	A timer endpoint
Twitter / camel-twitter `twitter://endpoint[?options]`	A twitter endpoint
Velocity / camel-velocity `velocity:someTemplateResource`	Generates a response using an Apache Velocity template

Component / ArtifactId / URI	Description
Vertx / camel-vertx `vertx:eventBusName`	Working with the vertx event bus
VM / camel-core `vm:name`	Asynchronous call to another endpoint in the same JVM
Weather / camel-weather `weather://name[?options]`	Polls the weather information from Open Weather Map.
Websocket / camel-websocket `websocket://hostname[:port][/resourceUri][?options]`	Communicating with Websocket clients.
WMQ / camel-wmq `wmq:type:name[?options]`	Communicating with IBM MQ without using the JMS wrapping. See Configuring connection to the WebSphere MQ native server using the WMQ component on page 130 for more information about the specific connection parameter and container configuration for IBM WebSphere MQ.
XQuery / camel-saxon `xquery:someXQueryResource`	Generates a response using an XQuery template
XSLT / camel-spring `xslt:someTemplateResource`	Generates a response using an XSLT template
Yammer / camel-yammer `yammer://function[?options]`	Allows you to interact with the Yammer enterprise social network.
Zookeeper / camel-zookeeper `zookeeper://zookeeperServer[:port][/path][?options]`	Working with ZooKeeper cluster(s)

Configuring connection to the WebSphere MQ native server using the WMQ component

When using the **cMessagingEndpoint** to address an endpoint in a WebSphere MQ native server by calling WMQ, the connection to the MQ QueueManager can be configured in the URI field or in a properties file. The following properties can be configured:

- `queueManagerName`: The name of the MQ QueueManager. If not specified, the component falls back to `default`.
- `queueManagerHostname`: The hostname of the MQ QueueManager.
- `queueManagerPort`: The port of the MQ QueueManager.
- `queueManagerChannel`: The channel of the MQ QueueManager.
- `queueManagerUserID`: The user ID (optional, only required for authentication).

- `queueManagerPassword`: The user password (optional, only required for authentication).
- `queueManagerCCSID`: The CCSID (optional, only required for authentication).

If the `queueManagerHostname`, `queueManagerPort`, and `queueManagerChannel` is not specified in the URI, the component loads a `mq.properties` file from the classloader. An example of a `mq.properties` shown as follows:

```
default.hostname=localhost
default.port=7777
default.channel=QM_TEST.SVRCONN
```

The `mq.properties` can contain multiple MQ Queue Managers definition. The format is:

```
name.hostname
name.port
name.channel
```

where the `name` is the QueueManager name. For example, the `mq.properties` file can contain:

```
default.hostname=localhost
default.port=7777
default.channel=DEFAULT.SVRCONN
test.hostname=localhost
test.port=7778
test.channel=QM_TEST.SVRCONN
```

The `mq.properties` also supports the `userID`, `password`, and `CCSID` properties. For example:

```
default.hostname=localhost
default.port=7777
default.channel=DEFAULT.SVRCONN
default.userID=mqm
default.password=mqm
default.CCSID=1208
```

To call the `mq.properties`, use a **cConfig** component and add it to the **Dependencies** table. To run the Route with this component in the studio, you need to download the *com.ibm.mq.jar*, *com.ibm.mq.commonservices.jar*, *com.ibm.mq.headers.jar*, *com.ibm.mq.jmqi.jar* and *connector.jar* from the IBM web site and add them to the **Dependencies** list of the **cConfig** too. For more information about the **cConfig** component, see cConfig properties on page 39.

If the Route with this component is deployed in Runtime, the `mq.properties` file will be called from `<TalendRuntimePath>/container/etc` folder. Furthermore, you need to download *com.ibm.mq.osgi.java_7.5.0.5.jar* from the IBM web site and add it to the `<TalendRuntimePath>/container/deploy` folder. Alternatively, copy the *com.ibm.mq.jar*, *com.ibm.mq.commonservices.jar*, *com.ibm.mq.headers.jar*, *com.ibm.mq.jmqi.jar* and *connector.jar* to the `<TalendRuntimePath>/container/lib/ext` folder and change `<TalendRuntimePath>/container/etc/custom.properties` by adding the MQ packages to `org.osgi.framework.system.packages.extra`:

```
org.osgi.framework.system.packages.extra = \
   com.ibm.mq; \
   com.ibm.mq.constants; \
   com.ibm.mq.exits; \
   com.ibm.mq.headers; \
   com.ibm.mq.headers.pcf; \
   com.ibm.mq.jmqi; \
   com.ibm.mq.pcf; \
   ...
```

For more information about the WMQ component, see the site https://github.com/camel-extra/camel-extra/tree/master/components/camel-wmq.

You can also use the **cMQConnectionFactory** component to create a connection to the WebSphere MQ native server, and use the **cWMQ** to communicate with the MQ QueueManager. For more information, see cMQConnectionFactory properties on page 146 and cWMQ properties on page 217.

cMessagingEndpoint Standard properties

These properties are used to configure cMessagingEndpoint running in the Standard Job framework.

The Standard cMessagingEndpoint component belongs to the Core family.

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**. For more information about the commonly used Camel components, see Commonly used Camel components on page 122.
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. **Note:** All the transitive dependencies of this custom component should be included in the jar file.

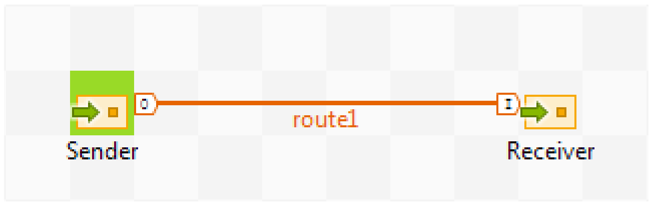
Usage

Usage rule	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 scenario applies only to a Talend solution with ESB.

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



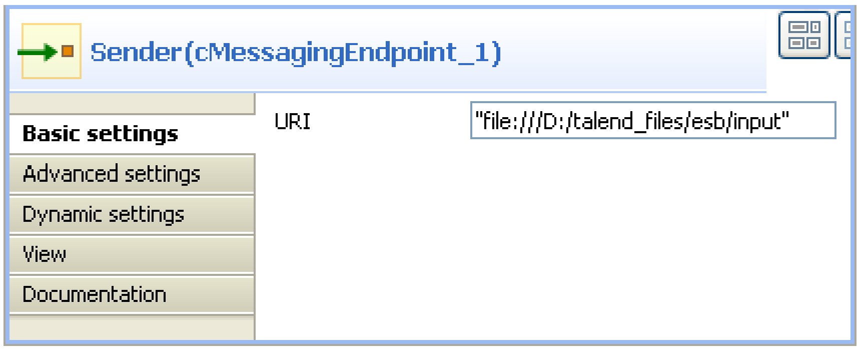
Dropping and linking the components

1. From the **Core** 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.

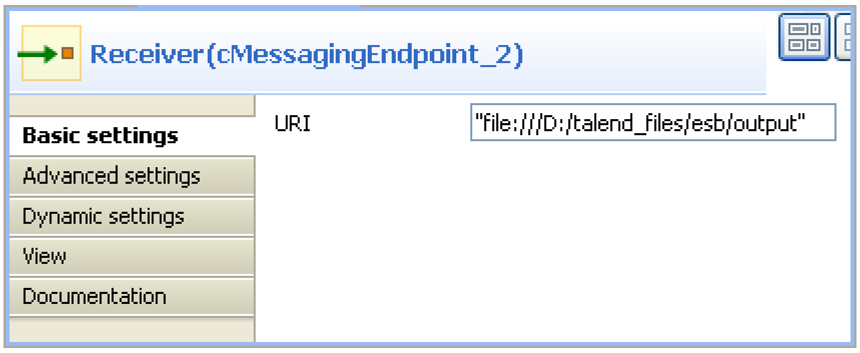
   As we are handling files, type in *"file:///"* and the path to the folder containing the files.



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.



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.

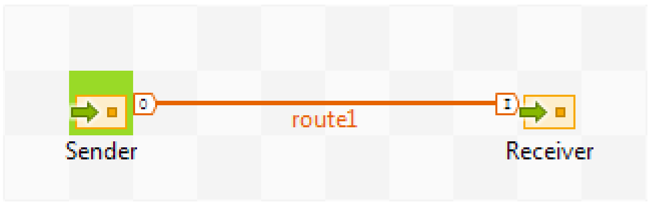
   You can also press **F6** to execute it.

   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 applies only to a Talend solution with ESB.

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



Dropping and linking components

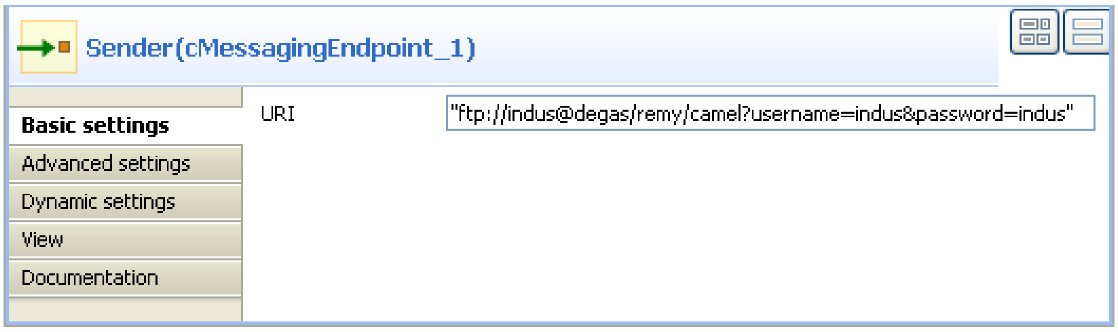
1. From the **Core** folder of the **Palette**, drag and drop two **cMessagingEndpoint** components onto the design workspace, one as the message sender and the other as the message receiver, and label them *Sender* and *Receiver* respectively to better identify their roles in the Route.

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

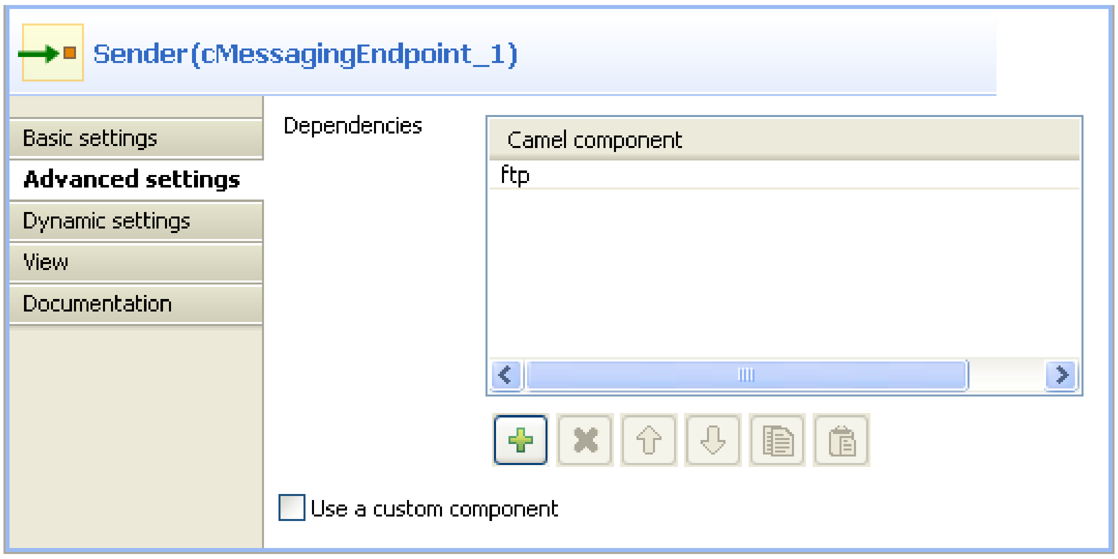
1. Double-click the component labeled *Sender* to display its **Basic settings** view in the **Component** tab.
2. In the **URI** field, type in the URI of the message you want to route.

   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.



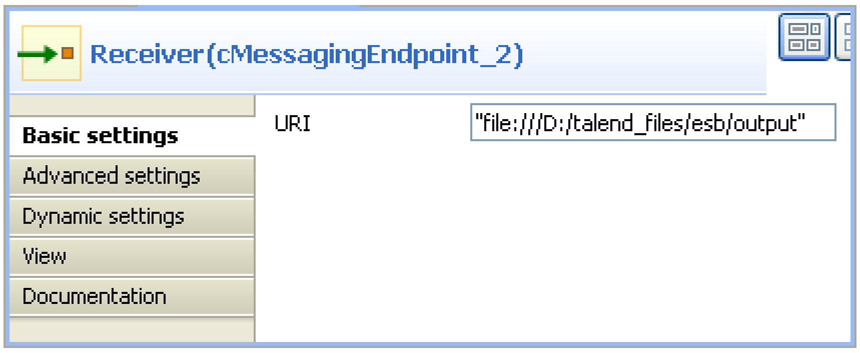
Dependencies

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.

Scenario 3: Using an Xquery endpoint to filter messages

This scenario applies only to a Talend solution with ESB.

In this scenario, we will use a **cMessagingEndpoint** component to call a Route Resource as an Xquery parser to extract messages from the local file system.

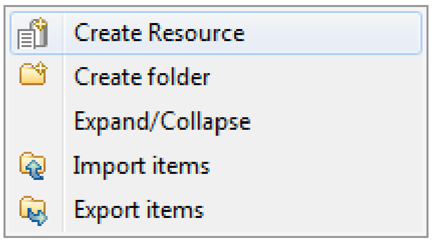
The following sample XML file is used in this scenario:

```
<people>
  <person id="8">
```

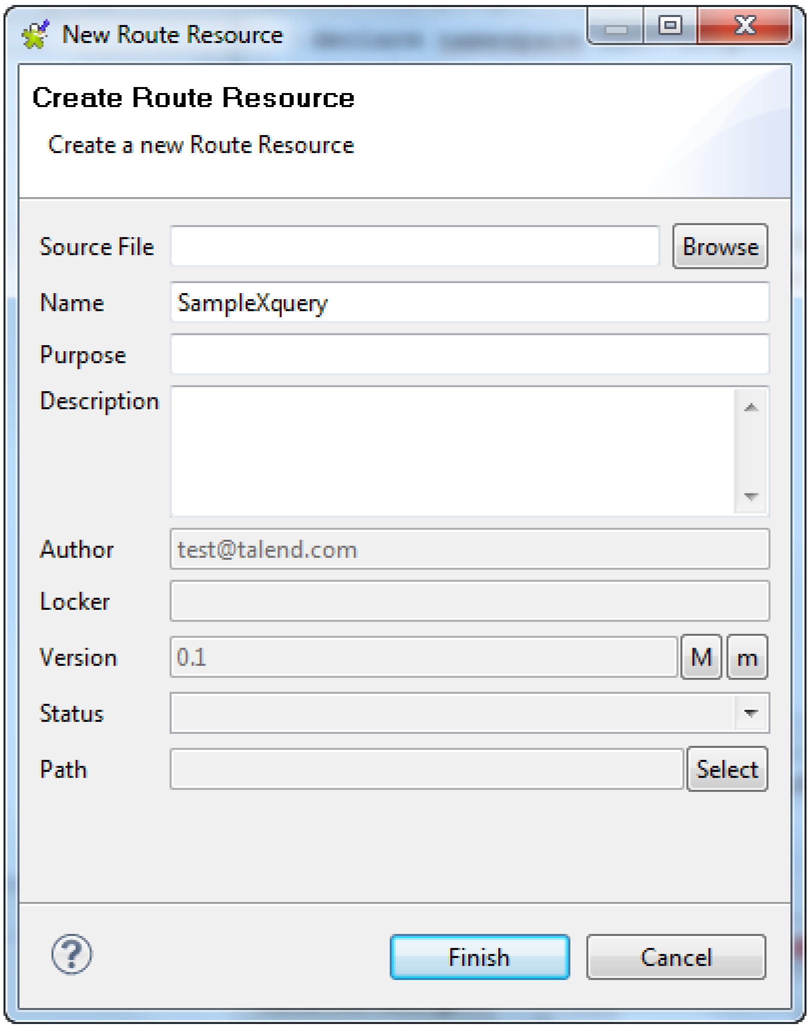
```
    <firstName>Ellen</firstName>
    <lastName>Ripley</lastName>
    <city>Washington</city>
  </person>
  <person id="9">
    <firstName>Peter</firstName>
    <lastName>Green</lastName>
    <city>London</city>
  </person>
</people>
```

Creating a Route Resource

1. From the repository tree view, right-click the **Resources** node and select **Create Resource** from the context menu.



2. The **[New Route Resource]** wizard opens. In the **Name** field, type in a name for the Resource, for example, *SampleXquery*. Click **Finish** to close the wizard.



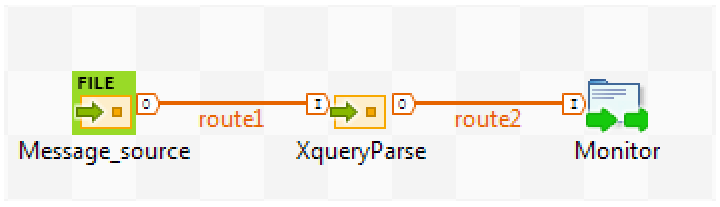
Source File:
Name: SampleXquery
Purpose:
Description:
Author: test@talend.com
Locker:
Version: 0.1
Status:
Path:

3. Enter the following code in the editor to extract the *firstName* and *lastName* of all the *person* elements.

```
declare namespace ns0="http://com.sap/b";
<people>
{
for $p in /people//person
return
        <person>
        <firstName>{$p/firstName/text()}</firstName>
        <lastName>{$p/lastName/text()}</lastName>
        </person>
}
</people>
```

4. Press **Ctrl+S** to save your Route Resource.

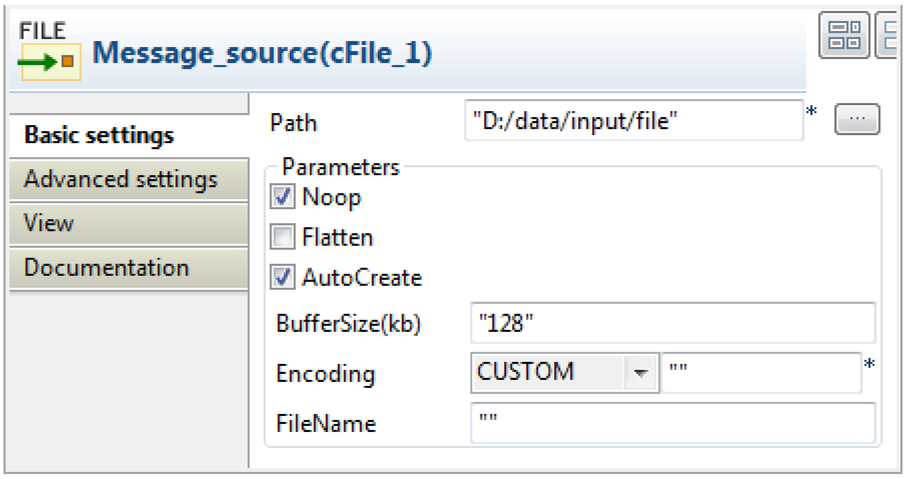
Dropping and linking the components



1. From the **Connectivity** folder of the **Palette**, drag and drop a **cFile** and a **cMessagingEndpoint** component onto the design workspace.
2. From the **Custom** folder, drag and drop a **cProcessor** component onto the design workspace.
3. Link the components with the **Row** > **Route** connection as shown above.
4. Label the components for better identification of their functionality.

Configuring the components and connections

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



Message_source(cFile_1)

Basic settings | Advanced settings | View | Documentation

Path: "D:/data/input/file"

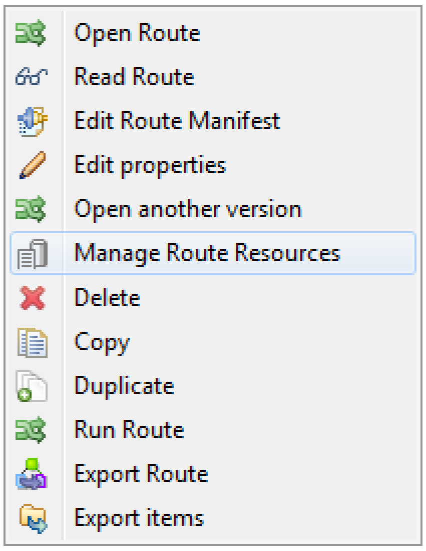
Parameters
- [✓] Noop
- [ ] Flatten
- [✓] AutoCreate

BufferSize(kb): "128"

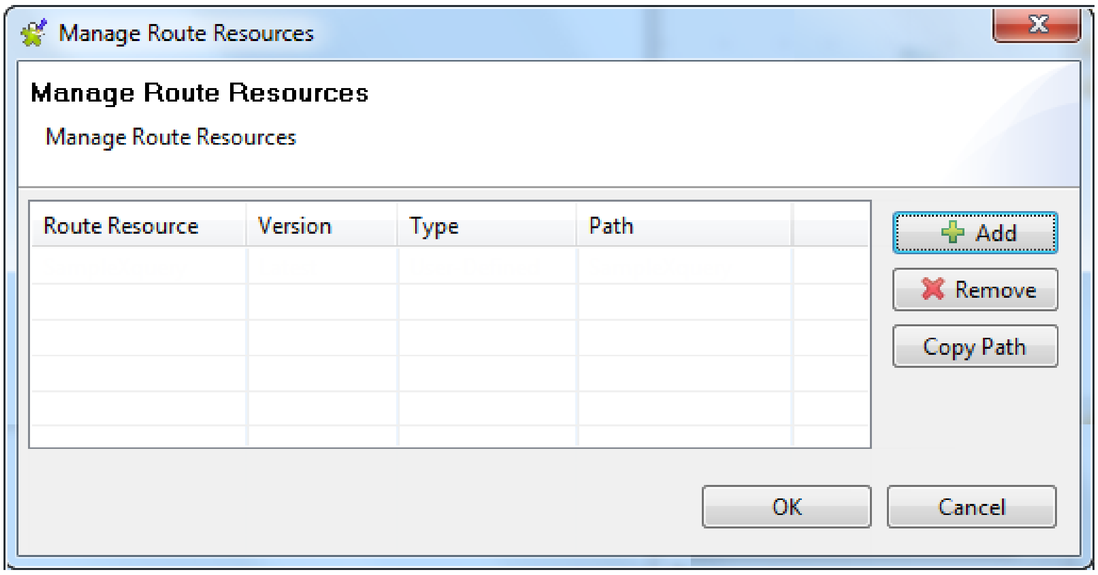
Encoding: CUSTOM ""

FileName: ""

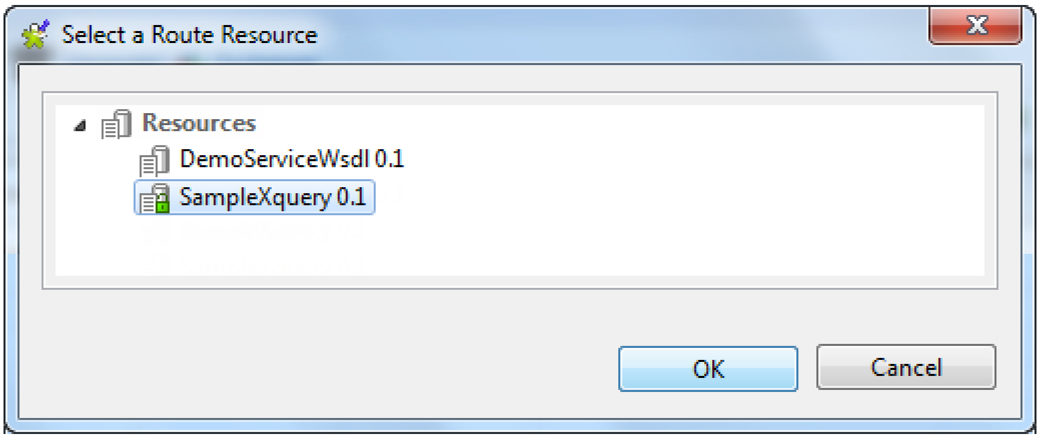
2. In the **Path** field, specify the path where the source file *people.xml* is located.
3. Right-click the Route from the repository tree view and select **Manage Route Resources** from the context menu.



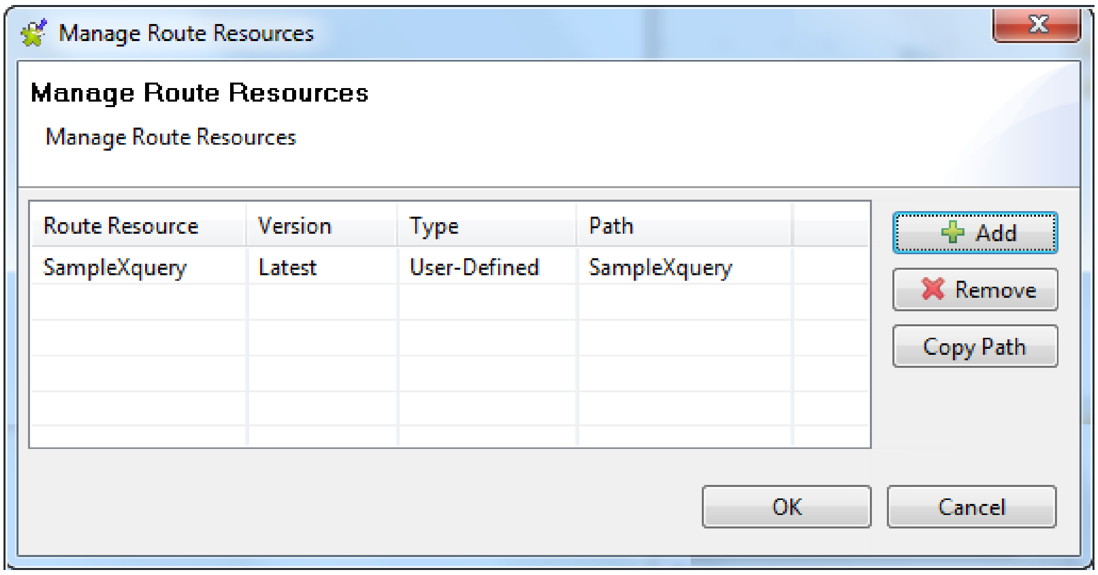
The **[Manage Route Resources]** wizard is opened.



4. In the **[Manage Route Resources]** wizard, click **Add** and select *SampleXquery* from the Resources tree view in the dialog. Click **OK**.

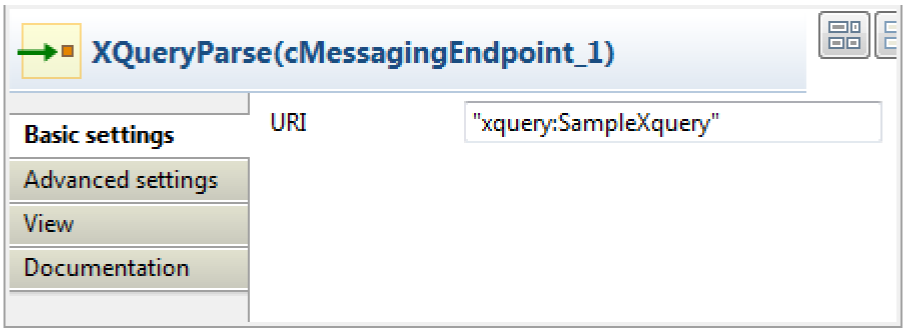


The *SampleXquery* Route Resource is added in the table of the **[Manage Route Resources]** wizard.



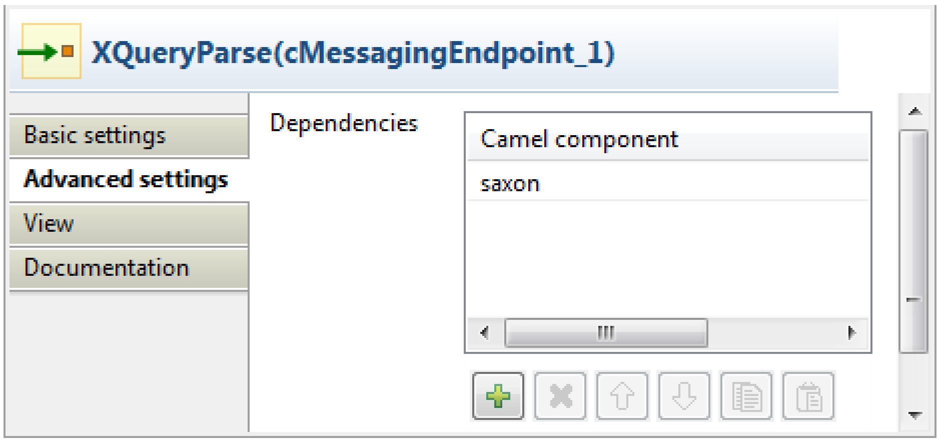
Route Resource	Version	Type	Path
SampleXquery	Latest	User-Defined	SampleXquery

5. Select the *SampleXquery* from the Route Resources list and click **Copy Path**. Click **OK** to close the wizard.
6. Double click the **cMessagingEndpoint** component to display its **Basic settings** view in the **Component** tab.



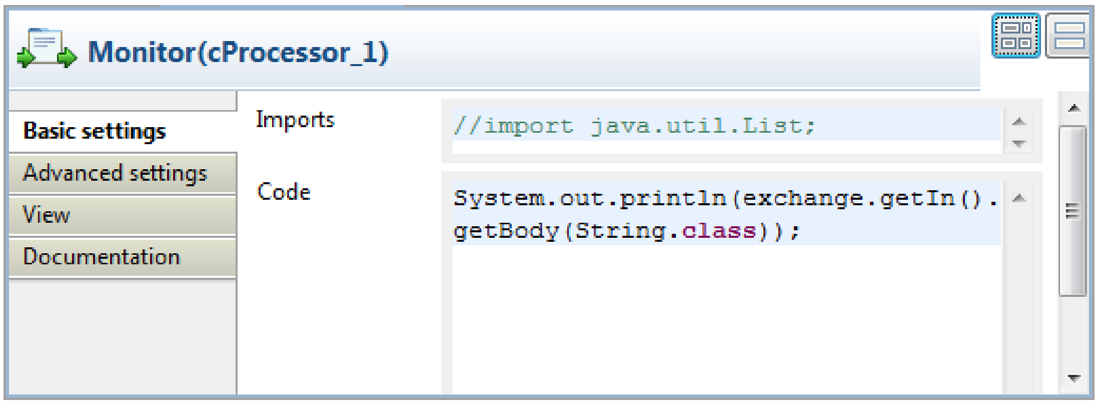
URI "xquery:SampleXquery"

7. In the **URI** field, enter *xquery:* and paste the path of the Route Resource *SampleXquery* that we just copied in double quotation marks.
8. Click the **Advanced settings** tab, add the Camel component **saxon** in the **Dependencies** list. For more information about **Xquery**, see Apache Camel's Website: http://camel.apache.org/xquery-endpoint.html.



Camel component
saxon

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



Imports
```
//import java.util.List;
```
Code
```
System.out.println(exchange.getIn().
getBody(String.class));
```

10. In the **Code** area, enter the following code to display the messages intercepted on the console:

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

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

Viewing code and executing the Route

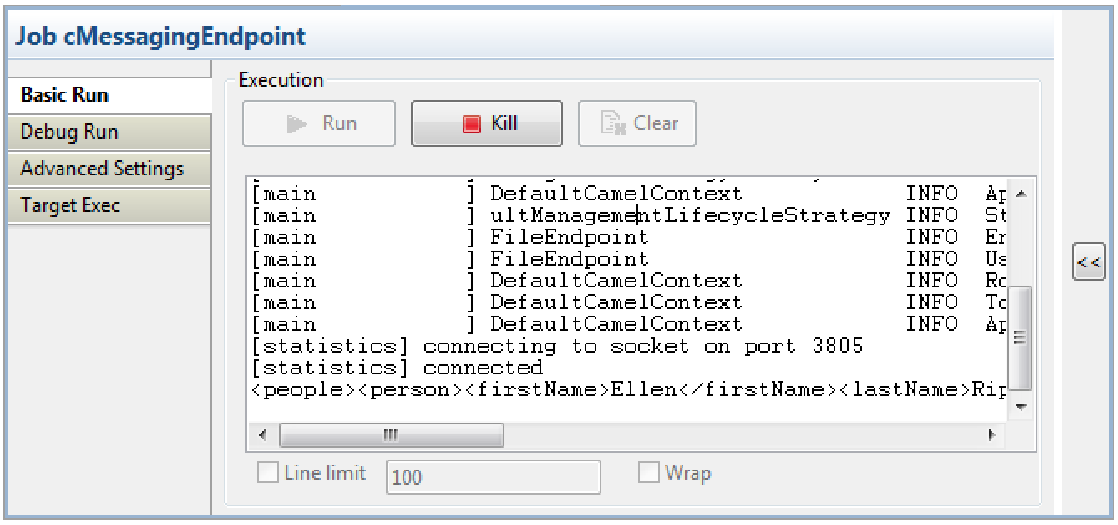
1. To have a look at the generated code, click the **Code** tab at the bottom of the design workspace.

```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Message_source_cFile_1"))
                    .routeId("Message_source_cFile_1")
                    .to(uriMap
                            .get("XQueryParse_cMessagingEndpoint_1"))
                    .id("cMessagingEndpoint_1")
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out
                                    .println("People in London:\n"
                                            + exchange
                                                    .getIn()
                                                    .getBody(
                                                            String.class));
                        }

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

As shown in the code above, the message is routed `from` `Message_source_cFile_1` `.to` `cMessagingEndpoint_1` and then processed by `cProcessor_1`.

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 *firstName* and *lastName* of all the *person* elements of the source file is printed in the console.



```
[main             ] DefaultCamelContext           INFO  Ap
[main             ] ultManagementLifecycleStrategy INFO  St
[main             ] FileEndpoint                  INFO  Er
[main             ] FileEndpoint                  INFO  Us
[main             ] DefaultCamelContext           INFO  Ro
[main             ] DefaultCamelContext           INFO  To
[main             ] DefaultCamelContext           INFO  Ap
[statistics] connecting to socket on port 3805
[statistics] connected
<people><person><firstName>Ellen</firstName><lastName>Rip
```

cMock properties

cMock is used to test Routes using test cases.

cMock simulates message generation and message endpoints, allowing you to test your Routes during Continuous Integration development.

cMock Standard properties

These properties are used to configure cMock running in the Standard Job framework.

The Standard cMock component belongs to the Testing family.

Basic settings

Validate message count	Select this check box to test that the correct number of messages are received on each endpoint. Specify the expected message number in the **expected number** field.
Header / Validate message header	Select this check box to test that the correct message header is received on each endpoint.
Header / Use File	This option appears when the **Validate message header** check box is selected. Select this option to specify the expected message header from a file and type in the name and path of the file between double quotes in the **File name** field.
Header / Use Inline Table	This option appears when the **Validate message header** check box is selected. Select this option

	to specify the expected message header from the table. Add as many rows as needed and enter the name and value of the header in the table.
Body / Validate message bodies	Select this check box to test that the correct message bodies are received on each endpoint.
Body / Use File	This option appears when the **Validate message bodies** check box is selected. Select this option to specify the expected message bodies from a file and type in the name and path of the file between double quotes in the **File name** field.
Body / Use Inline Table	This option appears when the **Validate message bodies** check box is selected. Select this option to specify the expected message bodies from the table. Add as many rows as needed and enter the message bodies in the table.
Simulate	This option appears when the **cMock** is used as a message producer. Select this check box to simulate the message generation, and select the **cProcessor** component to produce the message in the **Use existing cProcessor** list.
Wait time (in millis)	This option appears when the **cMock** is used to check the test output result. Specify the time in milliseconds that the **cMock** waits for the result to come.

Usage

Usage rule	**cMock** can be a start, middle or end component in a Route.
Limitation	The **cMock** component is designed for testing Routes. It is not recommended to use the **cMock** in message routing. When you add **cMock** to a

	Route, each exchange sent to it will be stored (to allow for later validation) in memory until explicitly reset or the JVM is restarted. If you are sending high volume and/or large messages, this may cause excessive memory use. For more information about how to test Routes, see the relevant section in the *Talend Studio User Guide*.

Related scenarios

No scenario is available for the Standard version of this component yet.

cMQConnectionFactory properties

cMQConnectionFactory is used to create a connection to a MQ server that can be called by multiple **cJMS**, **cWMQ**, **cAMQP** or **cMQTT** components in a Route.

cMQConnectionFactory encapsulates a set of connection configuration parameters to connect to a MQ server. This component replaces the former **cJMSConnectionFactory** and provides some enhancements.

Configuring connection to the WebSphere MQ native server in a properties file

The connection to the WebSphere MQ native server can also be configured in a properties file. If the Hostname, Port, and Channel is not specified in the component fields, the component loads a `mq.properties` file from the classloader. An example of a `mq.properties` shown as follows:

```
default.hostname=localhost
default.port=7777
default.channel=QM_TEST.SVRCONN
```

The `mq.properties` can contain multiple MQ Queue Managers definition. The format is:

```
name.hostname
name.port
name.channel
```

where the `name` is the queue manager name. For example, the `mq.properties` file can contain:

```
default.hostname=localhost
default.port=7777
default.channel=DEFAULT.SVRCONN
test.hostname=localhost
test.port=7778
test.channel=QM_TEST.SVRCONN
```

The `mq.properties` also supports the optional `userID`, `password`, and `CCSID` properties, which are required only for authentication. For example:

```
default.hostname=localhost
default.port=7777
default.channel=DEFAULT.SVRCONN
default.userID=mqm
default.password=mqm
default.CCSID=1208
```

To call the `mq.properties`, add it to the **Dependencies** table.

If the Route with this component is deployed in Runtime, the component calls the `mq.properties` file from `<TalendRuntimePath>/container/etc` folder.

For more information about how to specify the connection to the WebSphere MQ native server in the component fields, see cMQConnectionFactory properties on page 146.

cMQConnectionFactory Standard properties

These properties are used to configure cMQConnectionFactory running in the Standard Job framework.

The Standard cMQConnectionFactory component belongs to the Connectivity/Messaging family.

Basic settings

MQ Server	Select an MQ server from ActiveMQ, WebSphere MQ Jms, WebSphere MQ Native, AMQP 1.0, MQTT, or Customized. The connection to the WebSphere MQ native server can also be configured in a properties file. For more information, see Configuring connection to the WebSphere MQ native server in a properties file on page 146.
Use Transaction (for **ActiveMQ**, **WebSphere MQ Jms**, and **Customized** only)	Select this check box to enable local transaction in the Route that consumes messages from the MQ server. If exception occurs in the Route, the message in the message broker will be sent to the dead letter queue after the maximumRedeliveries configured for the Redelivery Policy, and will not be consumed by the Route. For more information, see the site http://camel.apache.org/transactional-client.html.
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 for ActiveMQ`.

HTTP Transport (for **ActiveMQ** only)	Select this check box to enable the HTTP based connection to the ActiveMQ broker.
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 (in ms) (for **ActiveMQ** only)	Specify the maximum waiting time (in milliseconds) before the connection breaks. This field is available only when the **Use PooledConnectionFactory** check box is selected.
Expiry Timeout (in ms) (for **ActiveMQ** only)	Specify the time (in milliseconds) before the connection breaks since it is used for the first time. The default value is 60000. The expiry is disabled if 0 is specified. This field is available only when the **Use PooledConnectionFactory** check box is selected.
Host Name (for **WebSphere MQ Jms**, **WebSphere MQ Native**, **AMQP 1.0 and MQTT** only)	Type in the name or IP address of the host on which the IBM WebSphere MQ server or the MQTT broker is running. For **WebSphere MQ Jms**, **WebSphere MQ Native** and **AMQP 1.0**, the default is localhost. For **MQTT**, the default is `127.0.0.1`.
Port (for **WebSphere MQ Jms**, **WebSphere MQ Native**, **AMQP 1.0** and **MQTT** only)	Type in the port of the MQ server. For **WebSphere MQ Jms** and **WebSphere MQ Native**, the default is `1414`. For **AMQP 1.0**, the default is `5672`. For **MQTT**, the default is `1883`.
Transport Type (for **WebSphere MQ Jms** 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 Jms** only)	Type in the name of the queue manager, or specify the name of the IBM WebSphere MQ server to find a queue manager.
Channel (for **WebSphere MQ Jms** and **WebSphere MQ Native** only)	Specify the name of the channel through which the connection is established. For **WebSphere MQ Jms**, the default is `SYSTEM.DEF.SVRCONN`. For **WebSphere MQ Native**, the default is `channel.name`.
Name (for **WebSphere MQ Native** only)	Specify the name of the queue manager to which the connection is established.
Authentication (for **ActiveMQ**, **WebSphere MQ Jms**, **WebSphere MQ Native**, **AMQP 1.0**, and **MQTT** only)	Select this check box and provide the username and password for the MQ server to validate the access permission. To enter the password, click the **[...]** button next to the password field, and then in the pop-up dialog box enter the password between double quotes and click **OK** to save the settings. For **WebSphere MQ Native** server, provide the CCSID (Coded Character Set Identifier) in addition that defines a numeric ordering of characters. For more information about CCSID, see the site http://www-01.ibm.com/software/globalization/cdra/appendix_c.html.
Dependencies (for **WebSphere MQ Jms**, **WebSphere MQ Native** and **Customized** only)	Specify additional libraries required by the MQ broker.
Use SSL (for **AMQP 1.0** and **MQTT** only)	Select this check box to connect to the MQ server over the SSL protocol. For **MQTT**, specify the TrustStore file containing the list of certificates that the MQ server trusts and enter the password used to check the integrity of the TrustStore data.
Connect Attempts (for **MQTT** only)	The maximum number of attempts to establish an initial connection, `-1` by default to use unlimited attempts.
Reconnect Attempts (for **MQTT** only)	The maximum number of attempts to re-establish a connection after a failure, `-1` by default to use unlimited attempts.

Reconnect Delay (for **MQTT** only)	The time in milliseconds between attempts to re-establish an initial or failed connection, `10` by default.
Quality of Service (for **MQTT** only)	The MQTT Quality of Service to use for message exchanges. It can be one of **AtMostOnce**, **AtLeastOnce** or **ExactlyOnce**.
Connect Wait In Seconds (for **MQTT** only)	Delay in seconds that the component will wait for a connection to be established to the MQTT broker, `10` by default.
Disconnect Wait In Seconds (for **MQTT** only)	The number of seconds the component will wait for a valid disconnect from the MQTT broker, `5` by default.
Send Wait In Seconds (for **MQTT** only)	The maximum time the component will wait for a receipt from the MQTT broker to acknowledge a published message before throwing an exception, `5` by default.
Codes (for **Customized** only)	Write a piece of code to specify the MQ connection factory to be used for message handling.

Usage

Usage rule	**cMQConnectionFactory** cannot be added directly in a Route.
Limitation	Due to license incompatibility, one or more JARs required to use this component are not provided. You can install the missing JARs for this particular component by clicking the **Install** button on the **Component** tab view. You can also find out and add all missing JARs easily on the **Modules** tab in the **Integration** perspective of your studio. You can find more details about how to install external modules in Talend Help Center (https://help.talend.com). To use the **WebSphere MQ Native** server, you need to download the *com.ibm.mq.jar*,

	com.ibm.mq.commonservices.jar, *com.ibm.mq.headers.jar*, *com.ibm.mq.jmqi.jar* and *connector.jar* from the IBM web site and add them to the **Dependencies** list.

Related scenario:

For a related scenario, see Scenario 1: Sending and receiving a message from a JMS queue.

cMQTT properties

cMQTT is used for communicating with MQTT compliant message brokers.

cMQTT sends messages to, or consumes messages from MQTT compliant message brokers.

cMQTT Standard properties

These properties are used to configure cMQTT running in the Standard Job framework.

The Standard cMQTT component belongs to the Connectivity/Internet of Things and Connectivity/Messaging family.

Basic settings

ConnectionFactory	This option appears when **Use Connection Factory** is selected. Click **[...]** and select a connection factory to be used for handling messages.
Topic Name	Type in a name for the message topic in the message broker.

Advanced settings

Parameters	Set the optional parameters in the corresponding table. Click **[+]** as many times as required to add parameters to the table. Then click the corresponding value field and enter a value. See the site http://camel.apache.org/amqp.html for available options.

Usage

Usage rule	**cMQTT** can be a start, middle or end component in a Route. It has to be used with the **cMQConnectionFactory** component, which creates a connection to a MQ server. For more information about **cMQConnectionFactory**, see cMQConnectionFactory properties on page 146.
Limitation	Due to license incompatibility, one or more JARs required to use this component are not provided. You can install the missing JARs for this particular component by clicking the **Install** button on the **Component** tab view. You can also find out and add all missing JARs easily on the **Modules** tab in the **Integration** perspective of your studio. You can find more details about how to install external modules in Talend Help Center (https://help.talend.com).

Scenario: Sending messages to and receiving messages from an MQTT broker

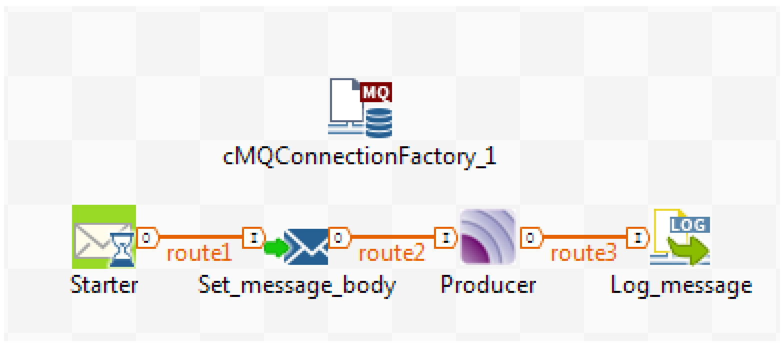
This scenario applies only to a Talend solution with ESB.

This scenario will show you how to use the **cMQTT** component to send messages to and consume messages from an MQTT broker. To do this, two Routes are built, a message producer Route, and a consumer Route. Messages are sent to the MQTT broker in the producer Route and then consumed in the consumer Route.

In this use case, Apache ActiveMQ is used as the message broker which supports the MQTT protocol. You need to launch the ActiveMQ server before executing the Route. For more information about installing and launching ActiveMQ server, see the site http://activemq.apache.org/index.html.

Building the producer Route

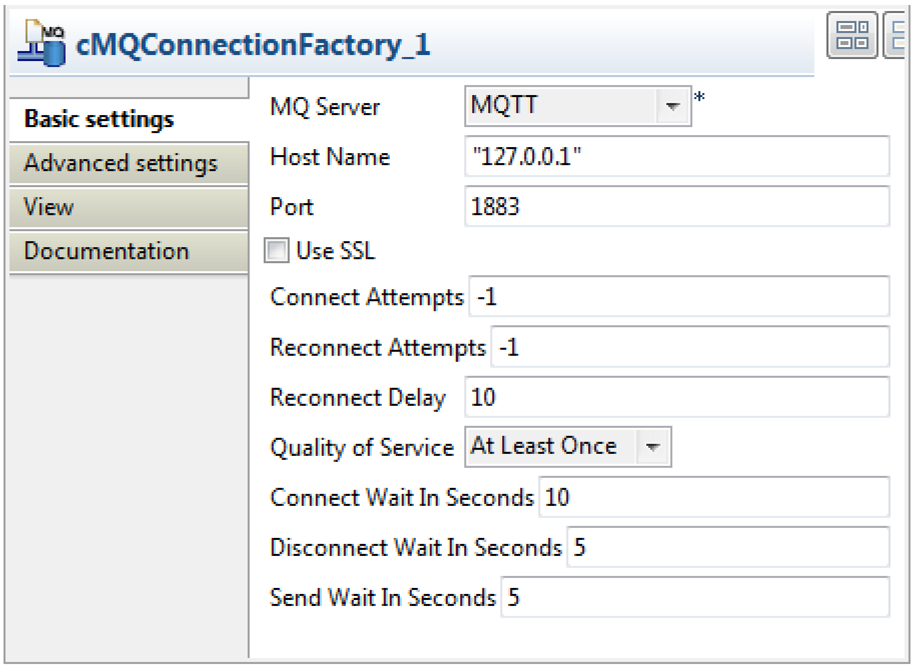
Dropping and linking the components



1. From the **Palette**, drag and drop a **cMQConnectionFactory**, a **cTimer**, a **cSetBody**, a **cMQTT**, and a **cLog** component onto the design workspace.
2. Label the components for better identification of their roles and link them with the **Row** > **Route** connection as shown above.

Configuring the components

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



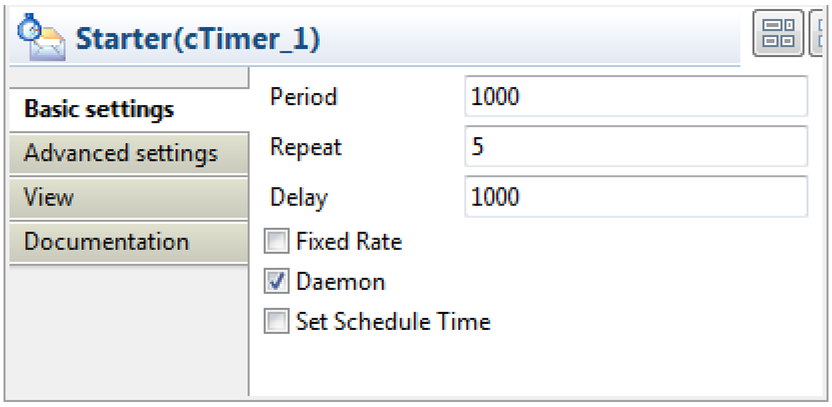
MQ Server: MQTT *
Host Name: "127.0.0.1"
Port: 1883
Use SSL: ☐
Connect Attempts: -1
Reconnect Attempts: -1
Reconnect Delay: 10
Quality of Service: At Least Once
Connect Wait In Seconds: 10
Disconnect Wait In Seconds: 5
Send Wait In Seconds: 5

2. From the **MQ Server** list, select **MQTT** to handle messages.

   In the **Host Name** field, keep the default value *"127.0.0.1"* as the local ActiveMQ server is used as the message broker in this example.

   In the **Port** field, keep the default *1883*.

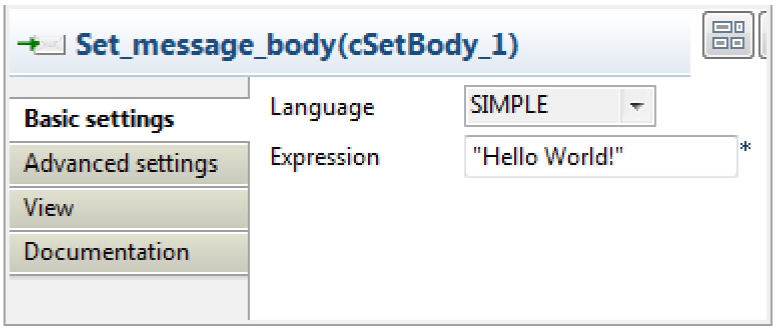
   Keep the default settings of the other options.
3. Double-click the **cTimer** component to open its **Basic settings** view in the **Component** tab.


Starter(cTimer_1)

Basic settings | Advanced settings | View | Documentation

Period: 1000
Repeat: 5
Delay: 1000
[ ] Fixed Rate
[✓] Daemon
[ ] Set Schedule Time

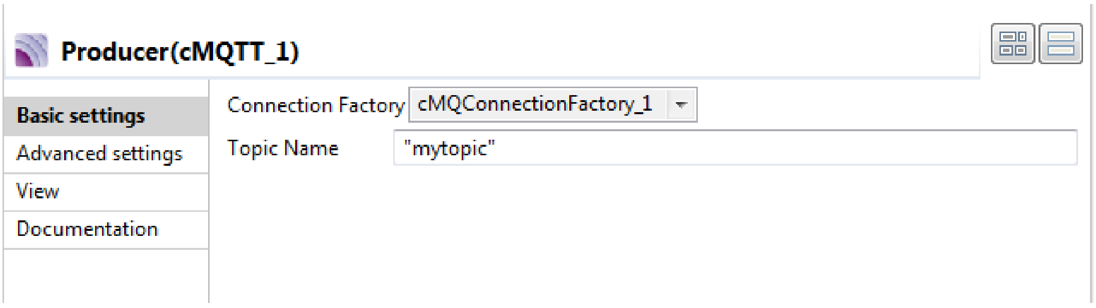
4. In the **Repeat** field, enter `5` to generate the message exchange five times. Keep the default settings of the other options.
5. Double-click the **cSetBody** component to open its **Basic settings** view in the **Component** tab.


Set_message_body(cSetBody_1)

Basic settings | Advanced settings | View | Documentation

Language: SIMPLE
Expression: "Hello World!" *

6. Select **SIMPLE** from the **Language** list box and type in `"Hello world"` in the **Expression** field as the message body.
7. Double-click the **cMQTT** component to open its **Basic settings** view in the **Component** tab.

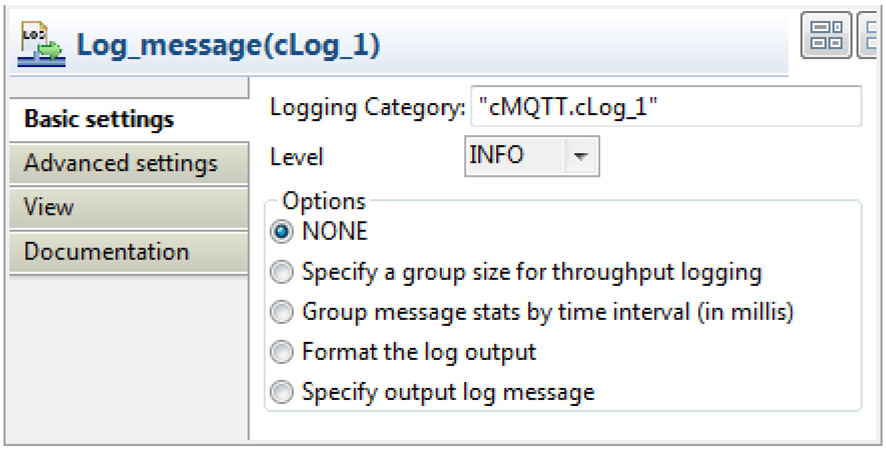

Producer(cMQTT_1)

Basic settings | Advanced settings | View | Documentation

Connection Factory: cMQConnectionFactory_1
Topic Name: "mytopic"

8. In the **ConnectionFactory** field, click **[...]** and select the MQ connection factory that you have just configured to handle messages.

   In the **Topic Name** field, type in a name for the topic, for example `"mytopic"`.
9. Keep the default settings of the **cLog** component to log the message exchanges.



Log_message(cLog_1)

Basic settings
Advanced settings
View
Documentation

Logging Category: "cMQTT.cLog_1"
Level: INFO

Options
- [✓ NONE]
- Specify a group size for throughput logging
- Group message stats by time interval (in millis)
- Format the log output
- Specify output log message

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

Viewing the code

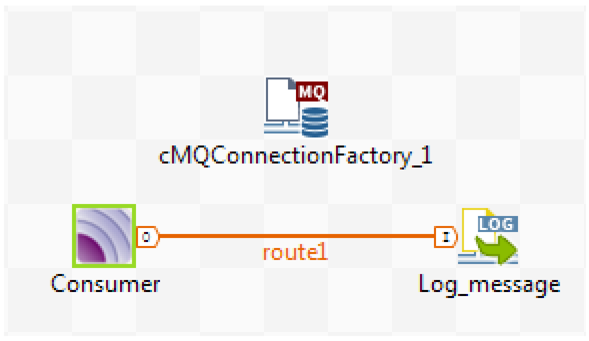
Click the **Code** tab at the bottom of the design workspace to check the generated code.

```
public void configure() throws java.lang.Exception {
    from(uriMap.get("cTimer_1")).routeId("Starter_cTimer_1").setBody()
            .simple("Hello World!").id("cSetBody_1")
            .to(uriMap.get("cMQTT_1")).id("cMQTT_1")
            .to(uriMap.get("cLog_1"))

            .id("cLog_1");
}
```

As shown above, the message flow from `cTimer_1` is given a payload by `cSetBody_1` and then sent to `cMQTT_1` and `cLog_1`.

Building the consumer Route

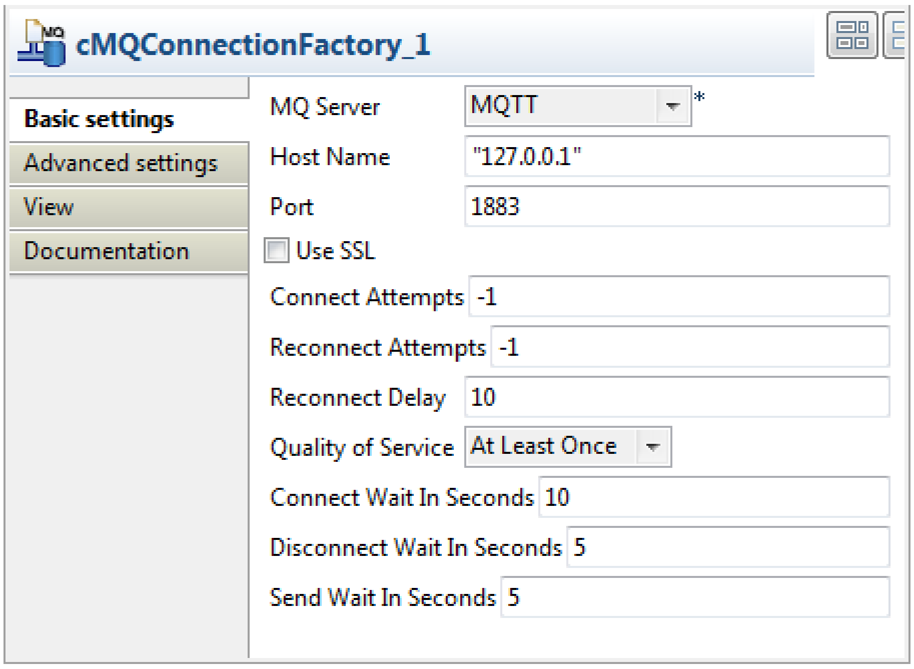
Arranging the flow of the message



1. From the **Palette**, drag and drop a **cMQConnectionFactory**, a **cMQTT**, and a **cLog** component onto the design workspace.
2. Label the components for better identification of their roles and link them with the **Row** > **Route** connection as shown above.

Configuring how the message is processed

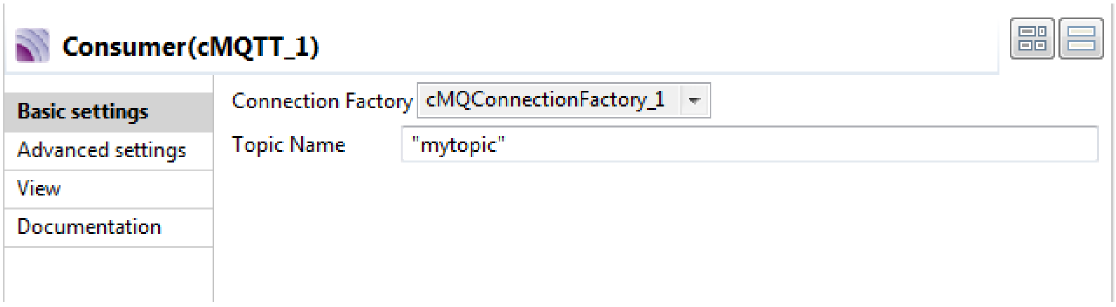
1. Double-click the **cMQConnectionFactory** component to display its **Basic settings** view in the **Component** tab.



cMQConnectionFactory_1

MQ Server	MQTT
Host Name	"127.0.0.1"
Port	1883
Use SSL	☐
Connect Attempts	-1
Reconnect Attempts	-1
Reconnect Delay	10
Quality of Service	At Least Once
Connect Wait In Seconds	10
Disconnect Wait In Seconds	5
Send Wait In Seconds	5

2. Configure the **cMQConnectionFactory** component as the one in the producer Route to connect to the same MQTT broker.
3. Double-click the **cMQTT** component to display its **Basic settings** view in the **Component** tab.

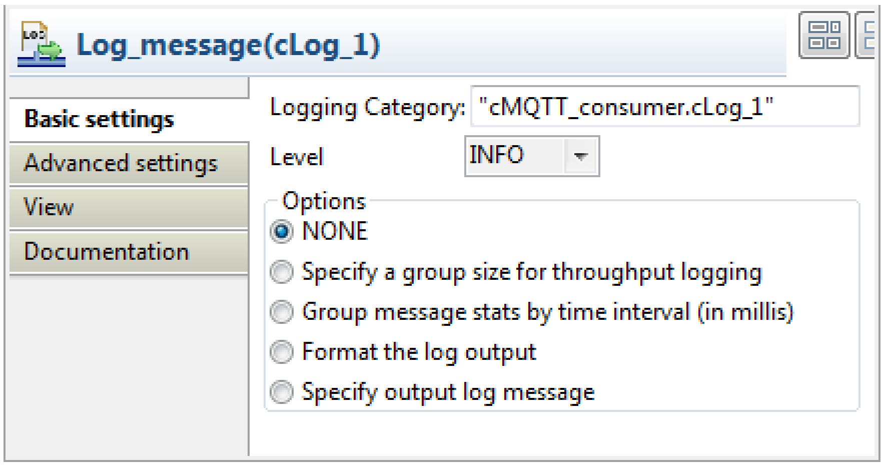


Consumer(cMQTT_1)

Connection Factory	cMQConnectionFactory_1
Topic Name	"mytopic"

4. In the **ConnectionFactory** field, click **[...]** and select the MQ connection factory that you have just configured to handle messages.

   Specify the same **Topic Name** in the consumer **cMQTT** component as in the producer.
5. Keep the default settings of the **cLog** component to log the message exchanges.



Logging Category: "cMQTT_consumer.cLog_1"
Level INFO
Options
[✓ NONE]
Specify a group size for throughput logging
Group message stats by time interval (in millis)
Format the log output
Specify output log message

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

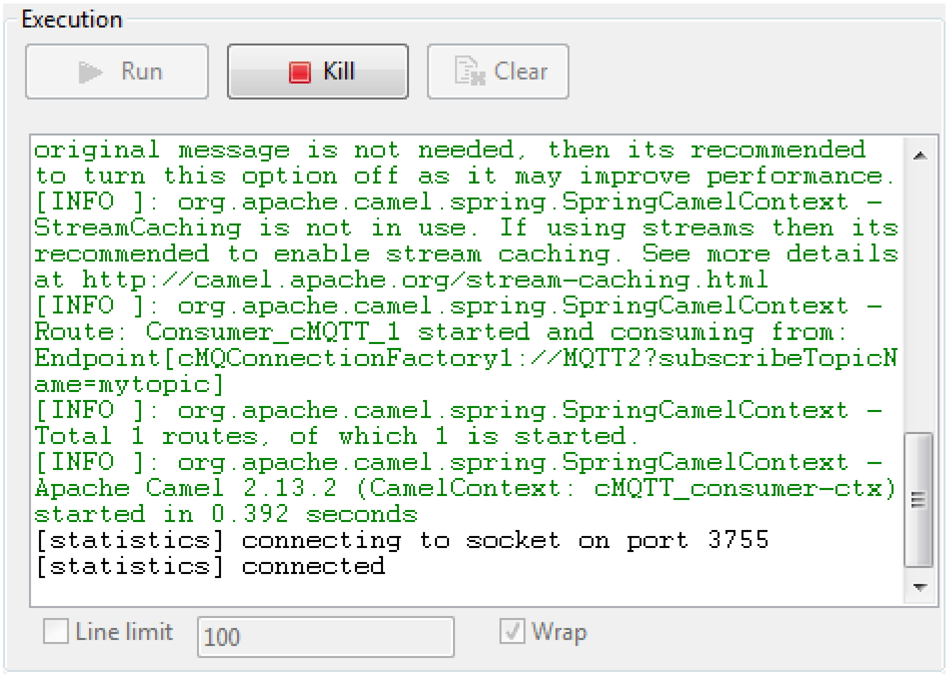
Viewing the code and executing the Route

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

```
public void configure() throws java.lang.Exception {
    from(uriMap.get("cMQTT_1")).routeId("Consumer_cMQTT_1")
            .to(uriMap.get("cLog_1"))

            .id("cLog_1");
}
```

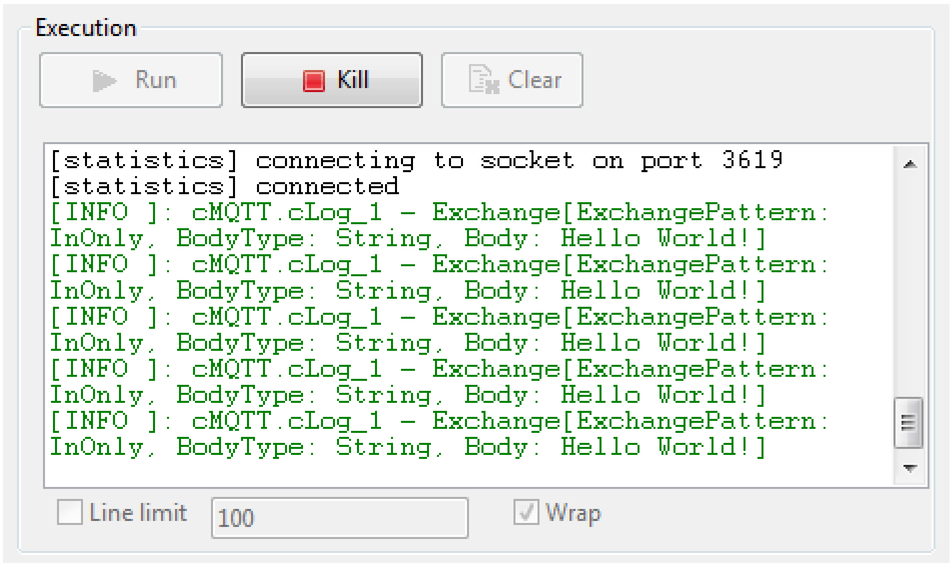
   As shown above, the message flow is routed from `cMQTT_1` to `cLog_1`.

2. Press **F6** to execute the Route. In the execution console you can see that there's no message exchange yet.



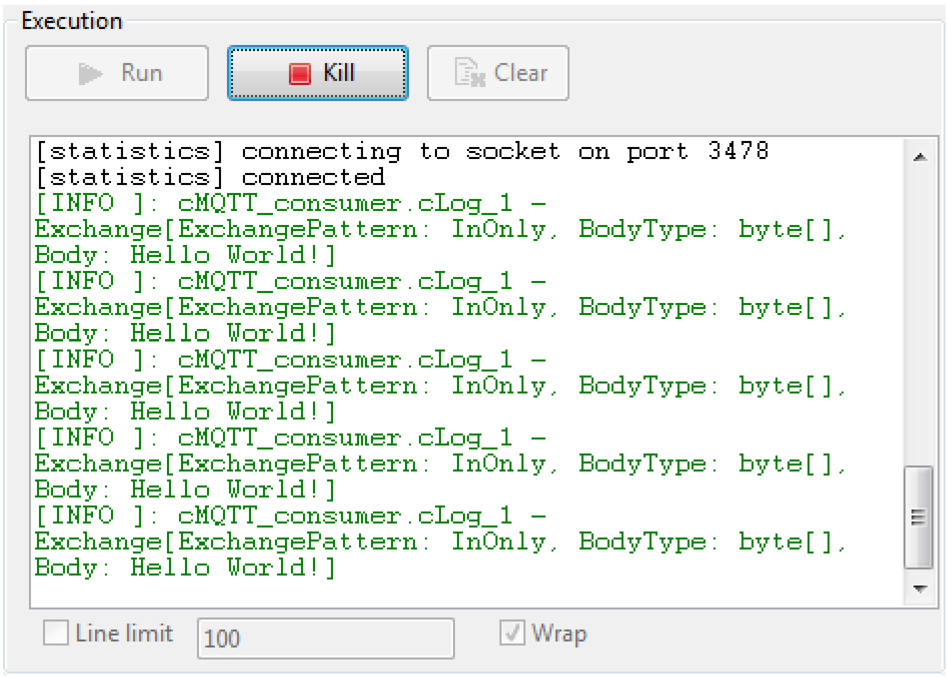
```
original message is not needed, then its recommended
to turn this option off as it may improve performance.
[INFO ]: org.apache.camel.spring.SpringCamelContext -
StreamCaching is not in use. If using streams then its
recommended to enable stream caching. See more details
at http://camel.apache.org/stream-caching.html
[INFO ]: org.apache.camel.spring.SpringCamelContext -
Route: Consumer_cMQTT_1 started and consuming from:
Endpoint[cMQConnectionFactory1://MQTT2?subscribeTopicN
ame=mytopic]
[INFO ]: org.apache.camel.spring.SpringCamelContext -
Total 1 routes, of which 1 is started.
[INFO ]: org.apache.camel.spring.SpringCamelContext -
Apache Camel 2.13.2 (CamelContext: cMQTT_consumer-ctx)
started in 0.392 seconds
[statistics] connecting to socket on port 3755
[statistics] connected
```

3. Execute the producer Route. The logs of the message exchange are printed in the console.



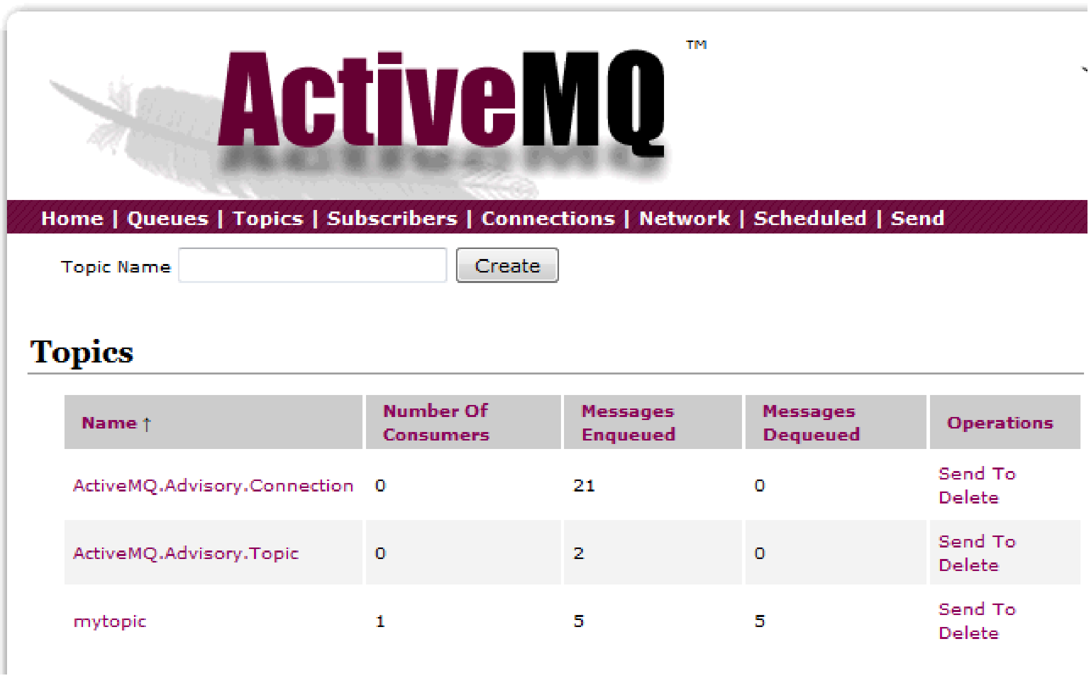
```
[statistics] connecting to socket on port 3619
[statistics] connected
[INFO ]: cMQTT.cLog_1 - Exchange[ExchangePattern:
InOnly, BodyType: String, Body: Hello World!]
[INFO ]: cMQTT.cLog_1 - Exchange[ExchangePattern:
InOnly, BodyType: String, Body: Hello World!]
[INFO ]: cMQTT.cLog_1 - Exchange[ExchangePattern:
InOnly, BodyType: String, Body: Hello World!]
[INFO ]: cMQTT.cLog_1 - Exchange[ExchangePattern:
InOnly, BodyType: String, Body: Hello World!]
[INFO ]: cMQTT.cLog_1 - Exchange[ExchangePattern:
InOnly, BodyType: String, Body: Hello World!]
```

4. In the consumer Route, the messages are consumed and shown in the execution console.



```
[statistics] connecting to socket on port 3478
[statistics] connected
[INFO ]: cMQTT_consumer.cLog_1 -
Exchange[ExchangePattern: InOnly, BodyType: byte[],
Body: Hello World!]
[INFO ]: cMQTT_consumer.cLog_1 -
Exchange[ExchangePattern: InOnly, BodyType: byte[],
Body: Hello World!]
[INFO ]: cMQTT_consumer.cLog_1 -
Exchange[ExchangePattern: InOnly, BodyType: byte[],
Body: Hello World!]
[INFO ]: cMQTT_consumer.cLog_1 -
Exchange[ExchangePattern: InOnly, BodyType: byte[],
Body: Hello World!]
[INFO ]: cMQTT_consumer.cLog_1 -
Exchange[ExchangePattern: InOnly, BodyType: byte[],
Body: Hello World!]
```

5. In the ActiveMQ Web Console, you can see that the topic *mytopic* has been created and the messages are consumed.


ActiveMQ™
Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send
Topic Name
Create

Topics

Name ↑	Number Of Consumers	Messages Enqueued	Messages Dequeued	Operations
ActiveMQ.Advisory.Connection	0	21	0	Send To Delete
ActiveMQ.Advisory.Topic	0	2	0	Send To Delete
mytopic	1	5	5	Send To Delete

cMulticast properties

cMulticast is used to route one or more messages to a number of endpoints at one go and process them in different ways.

cMulticast routes one or more messages to a number of endpoints at one go.

cMulticast Standard properties

These properties are used to configure cMulticast running in the Standard Job framework.

The Standard cMulticast component belongs to the Routing family.

Basic settings

URIS	Add as many lines as needed in the URIs table to define the endpoints to route the message(s) to.
Use ParallelProcessing	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 **Use ParallelProcessing** 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.
Stop On Exception	Select this check box to stop the processing immediately when an exception occurred.

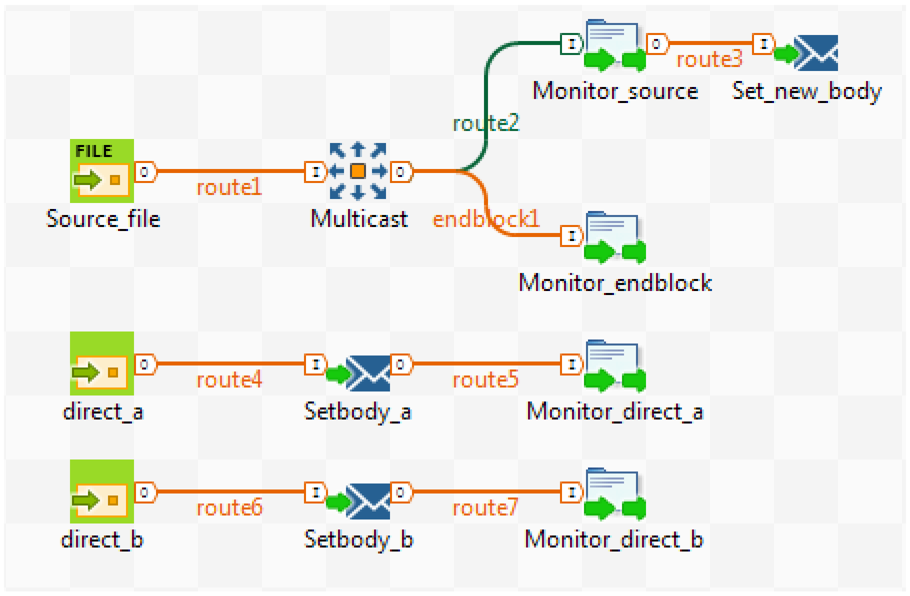
Usage

Usage rule	**cMulticast** can be used as a middle or end component in a Route.
Connections	**Route**
	EndBlock
Limitation	n/a

Scenario: Route a message to multiple endpoints and set a new body for each

This scenario applies only to a Talend solution with ESB.

In this scenario, a **cMulticast** component is used to route a message to two endpoints. The source message and the message on each endpoint is then set a new body. The **cProcessor** component is used to monitor the messages.

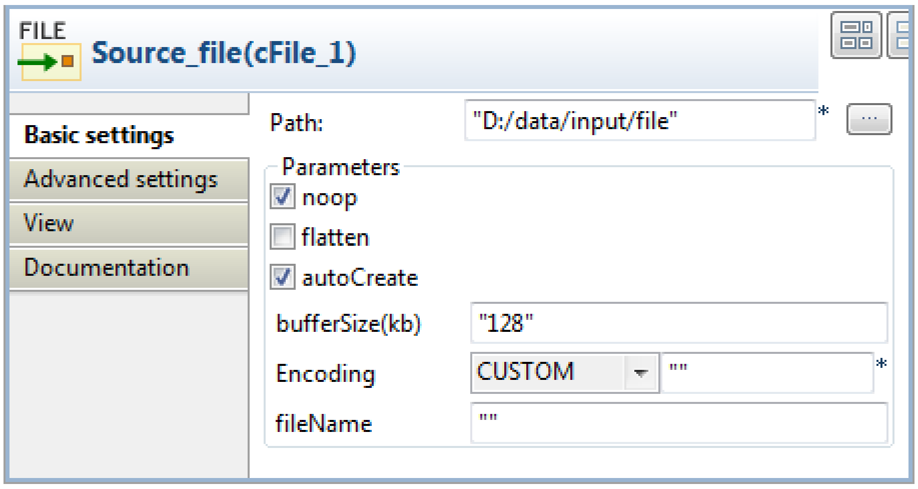


Dropping and linking the components

1. From the **Palette**, expand the **Connectivity** folder. Drag and drop a **cFile** and two **cMessagingEndpoint** components onto the design workspace.
2. From the **Routing** folder, drag and drop a **cMulticast** component onto the design workspace.
3. From the **Custom** folder, drag and drop four **cProcessor** components onto the design workspace.
4. From the **Core** folder, drag and drop three **cSetBody** components onto the design workspace.
5. Label the components as shown above to better identify their roles in the Route.
6. Right-click the **cMulticast** component, select **Row** > **EndBlock** in the context menu and click the **cProcessor** component labeled **Monitor_endblock**.
7. Right-click the **cFile** component, select **Row** > **Route** in the context menu and click the **cMulticast** component. Repeat this step to link the rest components in the Route as shown above using the **Row** > **Route** connection.

Configuring the components

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



Path: "D:/data/input/file"

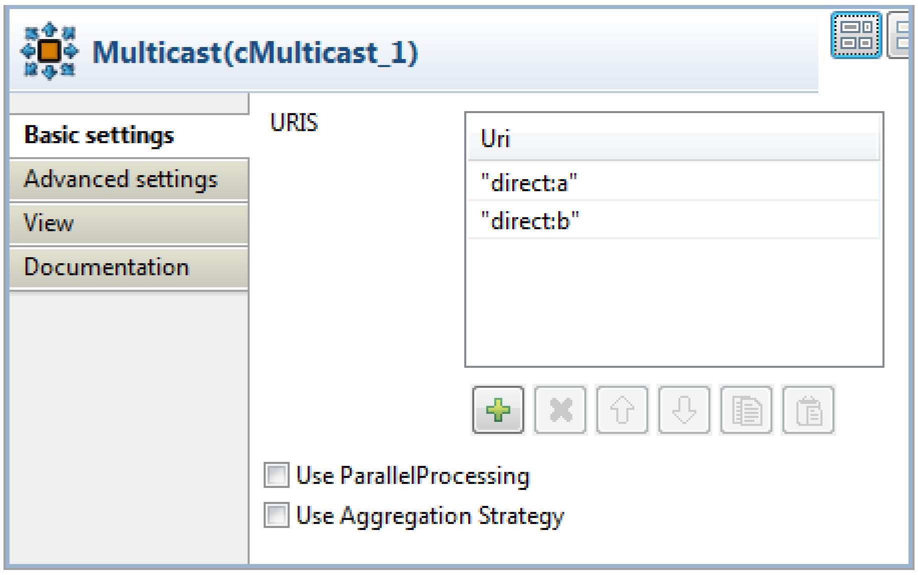
- [✓] noop
- [ ] flatten
- [✓] autoCreate

bufferSize(kb): "128"

Encoding: CUSTOM ""

fileName: ""

2. In the **Path** field, fill in or browse to the path where the source file *Hello.txt* is located. Keep the default settings for other fields.
3. Double-click the **cMulticast** component labeled *Multicast* to open its **Basic settings** view in the **Component** tab.

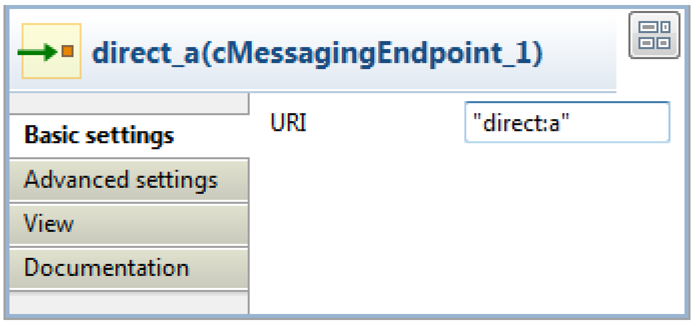


URIS

Uri
"direct:a"
"direct:b"

☐ Use ParallelProcessing
☐ Use Aggregation Strategy

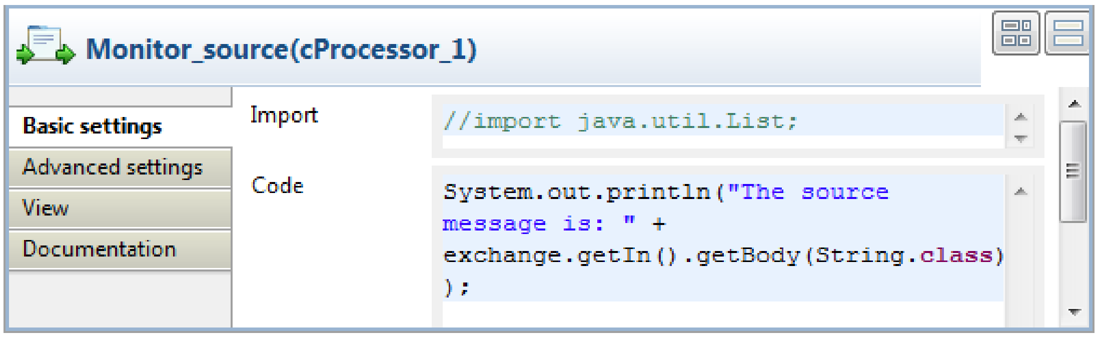
4. In the **URIS** table, click the plus button to add two lines and specify the URIs of the endpoints where the message will be sent, *"direct:a"* and *"direct:b"* in this use case.
5. Double-click the **cMessagingEndpoint** component labeled *direct_a* to open its **Basic settings** view in the **Component** tab.



URI "direct:a"

6. In the **URI** field, enter the endpoint URI, *"direct:a"* in this use case.

   Repeat this step to set the endpoint URI for *direct_b* as *"direct:b"*.
7. Double-click the **cProcessor** component labeled *Monitor_source* to open its **Basic settings** view in the **Component** tab.



Import

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

Code

```
System.out.println("The source
message is: " +
exchange.getIn().getBody(String.class)
);
```

8. In the **Code** box, enter the code below to print the source message in the console.

```
System.out.println("The source message is: " + exchange.getIn().getBody(String.class));
```

   Repeat this step to customize the code of **Monitor_endblock**, **Monitor_direct_a**, and **Monitor_direct_b** as shown below to print the message of each endpoint.

Monitor_endblock:

System.out.println("The endblock message is: " + exchange.getIn().getBody(String.class));

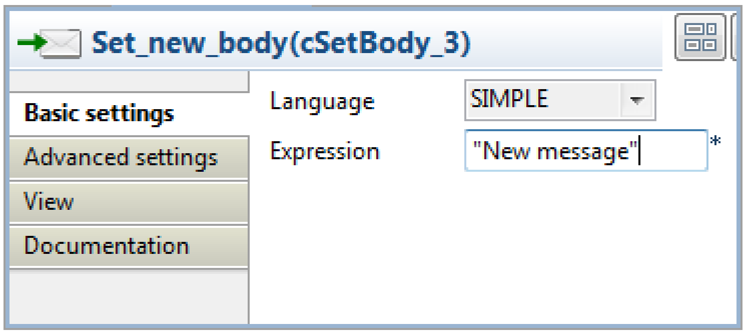
Monitor_direct_a:

System.out.println("direct a just downloaded: "+exchange.getIn().getBody(String.class));

Monitor_direct_b:

System.out.println("direct b just downloaded: "+exchange.getIn().getBody(String.class));

9. Double-click the **cSetBody** component labeled *Set_new_body* to open its **Basic settings** view in the **Component** tab.



Language: SIMPLE
Expression: "New message"

10. Select **SIMPLE** in the **Language** list.

    In the **Expression** field, enter *"New message"* as the new message body.

    Repeat this step to set the message body for *direct:a* and *direct:b* as *"message A"* and *"message B"* respectively.

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

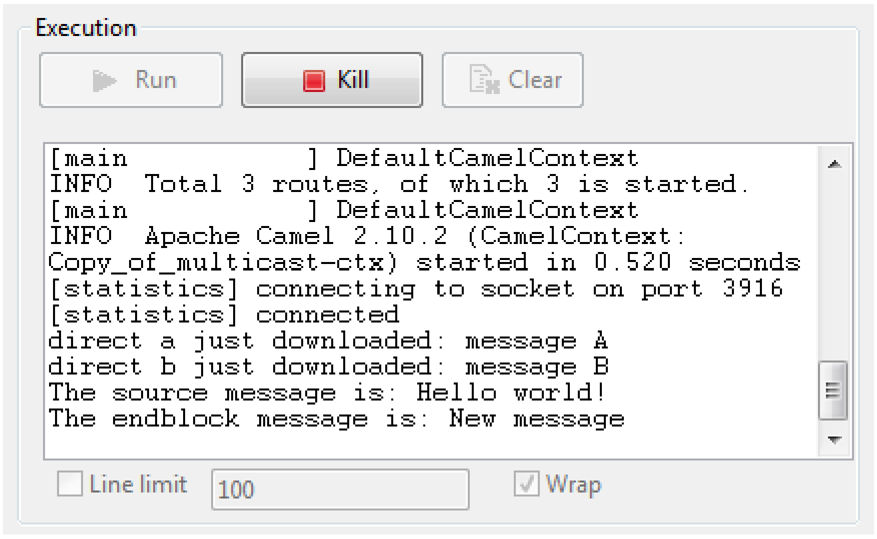
Viewing code and executing the Route

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

```
public RouteBuilder route() {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("Source_file_cFile_1"))
                    .routeId("Source_file_cFile_1")
                    .multicast()
                    .to("direct:a", "direct:b")
                    .id("cMulticast_1")
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out.println("The source message is: "
                                    + exchange.getIn()
                                            .getBody(String.class));
                        }

                    }).id("cProcessor_1").setBody().simple("New message")
                    .id("cSetBody_3").end()
                    .process(new org.apache.camel.Processor() {
                        public void process(
                                org.apache.camel.Exchange exchange)
                                throws Exception {
                            System.out.println("The endblock message is: "
                                    + exchange.getIn()
                                            .getBody(String.class));
                        }

                    }).id("cProcessor_2");
            from(uriMap.get("direct_a_cMessagingEndpoint_1"))
                    .routeId("direct_a_cMessagingEndpoint_1").setBody()
                    .simple("message A").id("cSetBody_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));
                        }

                    }).id("cProcessor_3");
```

In the partially shown code, the source message is routed `from "Source_file_cFile_1" .to direct:a` and `direct:b` via `"cMulticast_1"`. The message is then processed by `"cProcessor_1"` and given the message body `"New message"` by `"cSetBody_3"`. The `.end` block of the route is processed by `"cProcessor_2"`. The message `from "direct_a_cMessagingEndpoint_1"` is set the message body `"message A"` by `"cSetBody_1"` and processed by `"cProcessor_3"`. The message from `direct:b` is processed similarly.

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.



```
[main             ] DefaultCamelContext
INFO  Total 3 routes, of which 3 is started.
[main             ] DefaultCamelContext
INFO  Apache Camel 2.10.2 (CamelContext:
Copy_of_multicast-ctx) started in 0.520 seconds
[statistics] connecting to socket on port 3916
[statistics] connected
direct a just downloaded: message A
direct b just downloaded: message B
The source message is: Hello world!
The endblock message is: New message
```

RESULT: The source file message is *Hello world!*. The message routed to *direct:a* and *direct:b* is set the message body *message A* and *message B* respectively. The end block message of this Route is *New message* that is set by the component labeled *Set_new_body*.

cOnException properties

cOnException allows you to specify the error handling you require on an exception type basis.

cOnException catches the exceptions defined and triggers certain actions which are then performed on these exceptions and the message routing.

cOnException Standard properties

These properties are used to configure cOnException running in the Standard Job framework.

The Standard cOnException component belongs to the Exception Handling family.

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 asynchronous delayed redelivery. For details, go to http://camel.apache.org/exception-clause.html.

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.

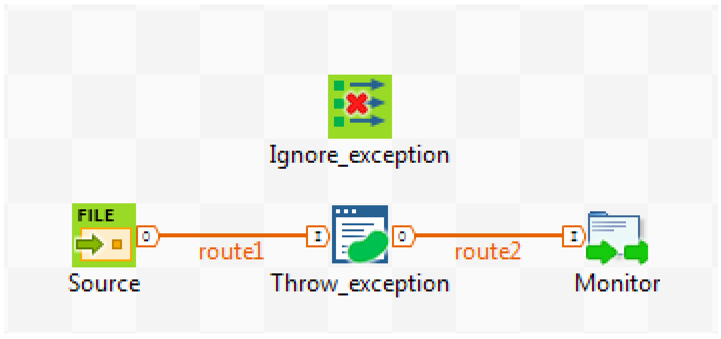
Usage

Usage rule	**cOnException** is used as a separate component in a Route.
Limitation	n/a

Scenario: Using cOnException to ignore exceptions and continue message routing

This scenario applies only to a Talend solution with ESB.

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.

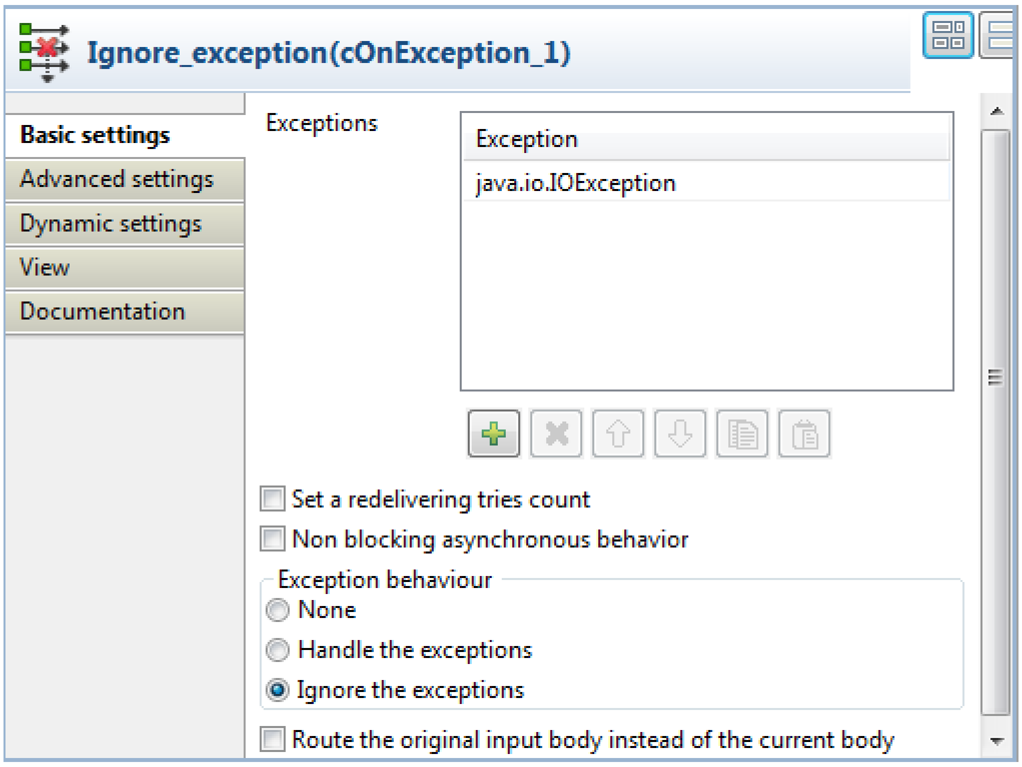


Dropping and linking the components

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

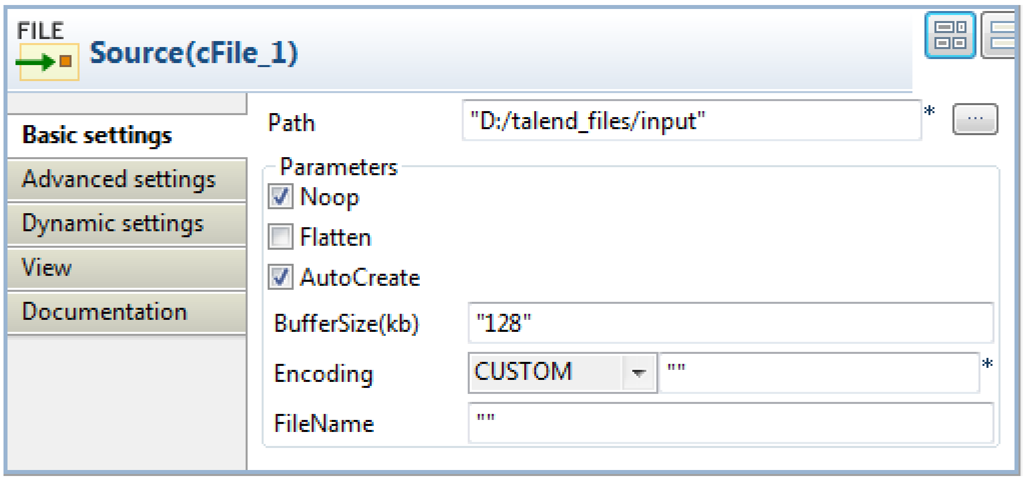
Exception behaviour: None [✓ Ignore the exceptions] Handle the exceptions

- [ ] 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.



Source(cFile_1)

Path "D:/talend_files/input"

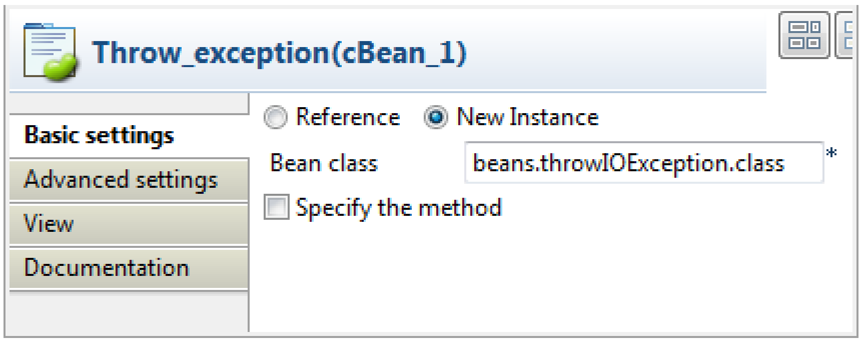
Parameters
- [x] Noop
- [ ] Flatten
- [x] AutoCreate

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.



Reference [✓ New Instance]

Bean class: beans.throwIOException.class *

Specify the method [ ]

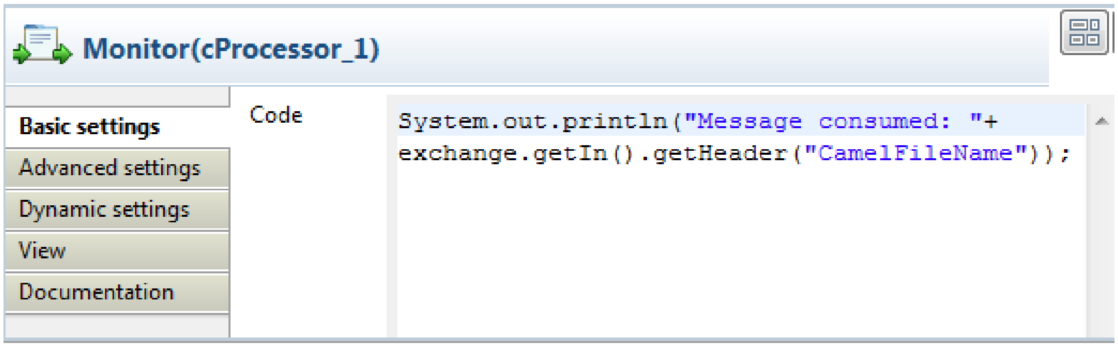
6. Select **New Instance** and in the **Bean class** field, enter the name of the bean to throw an IO exception, *beans.throwIOException.class* in this scenario.

   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 Studio 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

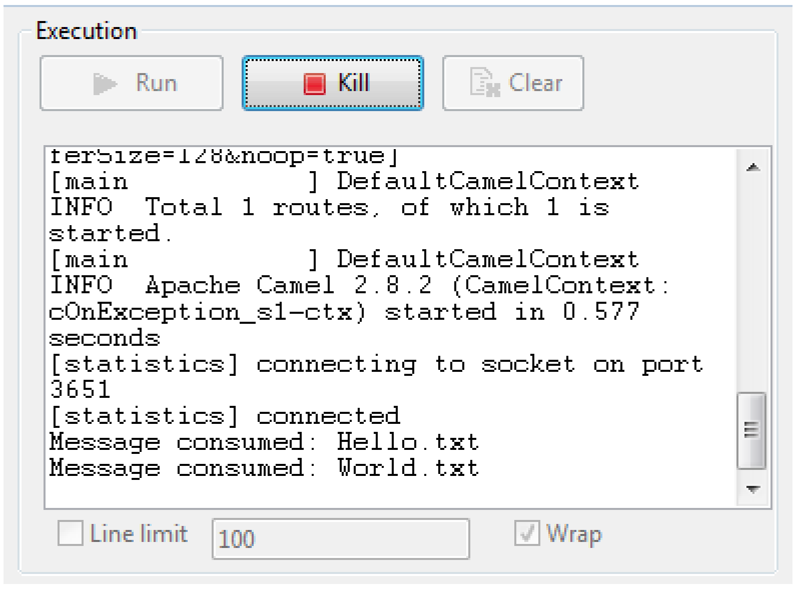
1. Click the **Code** tab at the bottom of the design workspace to check the generated code.

   ```
   }

   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.

cPipesAndFilters properties

This component allows you to split message routing into a series of independent processing stages.

The **cPipesAndFilters** component divides message processing into a sequence of independent endpoint instances, which can then be chained together.

cPipesAndFilters Standard properties

These properties are used to configure cPipesAndFilters running in the Standard Job framework.

The Standard cPipesAndFilters component belongs to the Routing family.

Basic settings

URI list	Click the plus button to add new lines for URIs that identify endpoints.

Usage

Usage rule	**cPipesAndFilters** is usually used in the middle of a Route.
Limitation	n/a

cProcessor properties

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.

cProcessor implements consumers of message exchanges or implements a Message Translator.

cProcessor Standard properties

These properties are used to configure cProcessor running in the Standard Job framework.

The Standard cProcessor component belongs to the Custom family.

Basic settings

Imports	Enter the Java code that helps to import, if necessary, external libraries used in the **Code** box.
Code	Type in the Java code you want to implement.

Usage

Usage rule	**cProcessor** is used as a middle or end component in a Route.
Limitation	n/a

Related scenario:

For a related scenario, see Scenario: Intercepting several routes and redirect them in a single new route of cIntercept properties on page 97.

cRecipientList properties

cRecipientList allows you to route messages to a number of dynamically specified recipients.

cRecipientList routes messages to a list of user defined recipients. It can also process the message before sending it to the recipients and assemble the replies from the sub-messages into a single outgoing message.

cRecipientList Standard properties

These properties are used to configure cRecipientList running in the Standard Job framework.

The Standard cRecipientList component belongs to the Routing family.

Basic settings

Language	Select the expression language from **None**, **Bean**, **Constant**, **CorrelationID**, **EL**, **Groovy**, **Header**, **JavaScript**, **JoSQL**, **JSonPath**, **JXPath**, **MVEL**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**.
Expression	Type in the expression that returns multiple endpoints.
Use Result Class Type	This option appears when **XPath** is selected in the **Language** list. Select this check box to set the result type of the sub-messages in the field that appears.
Add Namespaces	This option appears when **XPath** is selected in the **Language** list. Select this check box to add namespaces for the Xpath expression. Click **[+]** to add as many namespaces as required to the table and define the prefix and URI in the corresponding columns.
Use Delimiter	Select this check box to customize the separator for the **Expression**. Enter the characters, strings or regular expressions to be used as the separator in the **Delimiter** field.
Use Strategy	Select this check box to refer to an aggregation strategy to assemble the replies from the sub-messages into a single outgoing message from the recipient list. Enter the ID of the aggregation strategy in the field.
Parallel Processing	Select this check box to send the message to the recipients simultaneously.
Use ExecutorService	This option appears when **Parallel Processing** is enabled. Select this check box to use a custom

	thread pool for parallel processing. Specify the thread pool in the **ExecutorService** field.
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.
Streaming	Select this check box to process the sub-message replies in the order that the replies are received from each recipient. If this option is disabled, the replies will be processed in the same order as specified by the **Expression**.
Timeout	Specify a total timeout in millisecond. If the message is not routed to the recipients and processed within the given time frame, the timeout triggers and the recipient list breaks out.
Use On-Prepare Processor	Select this check box to use a custom processor to prepare the copy of the exchange that each recipient will receive. Enter the ID of the processor in the next field. You can use the **cBeanRegister** to register a Java bean as a processor.
Share Unit of Work	Select this check box to share the unit of work between the parent exchange and each recipient exchange. See the same option of **cSplitter** for more information.

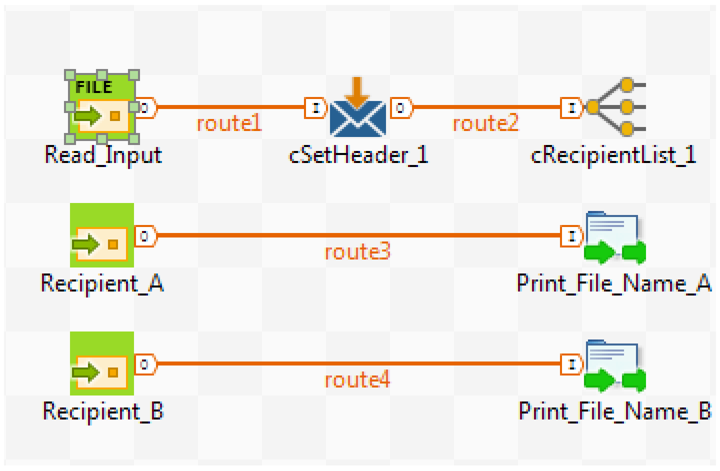
Usage

Usage rule	**cRecipientList** is used as a middle component in a Route.
Limitation	n/a

Scenario: Routing a message to multiple recipients

This scenario applies only to a Talend solution with ESB.

In this scenario, a **cRecipientList** component is used to route a message to a list of recipients.



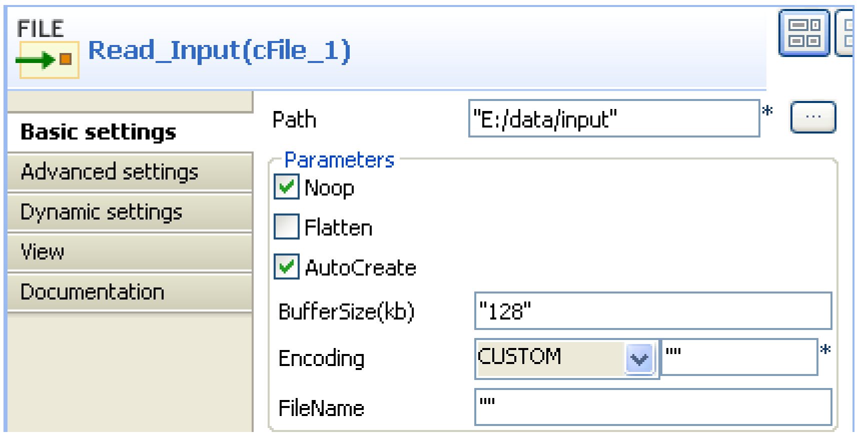
To build the Route, do the following.

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"

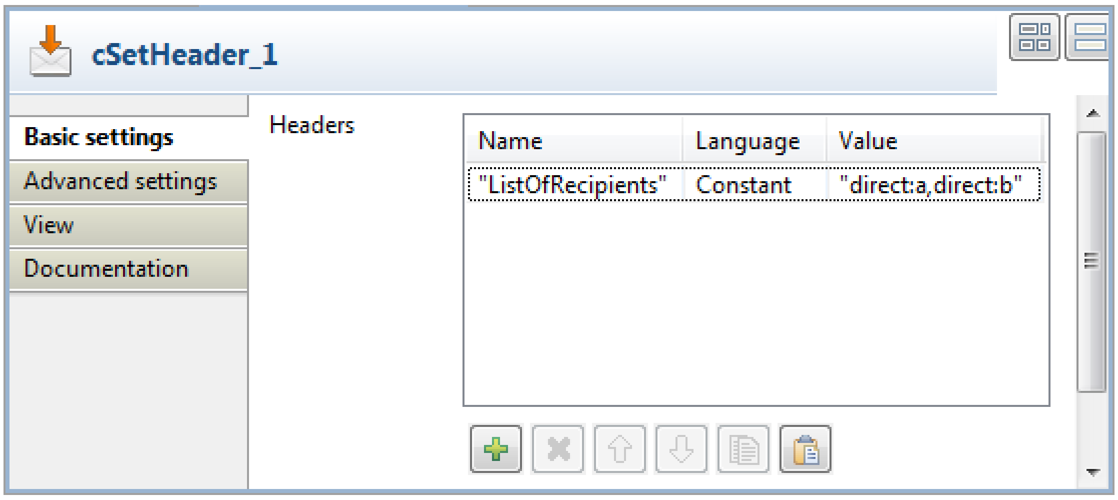
- [✓] Noop
- [ ] Flatten
- [✓] AutoCreate

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.



Headers

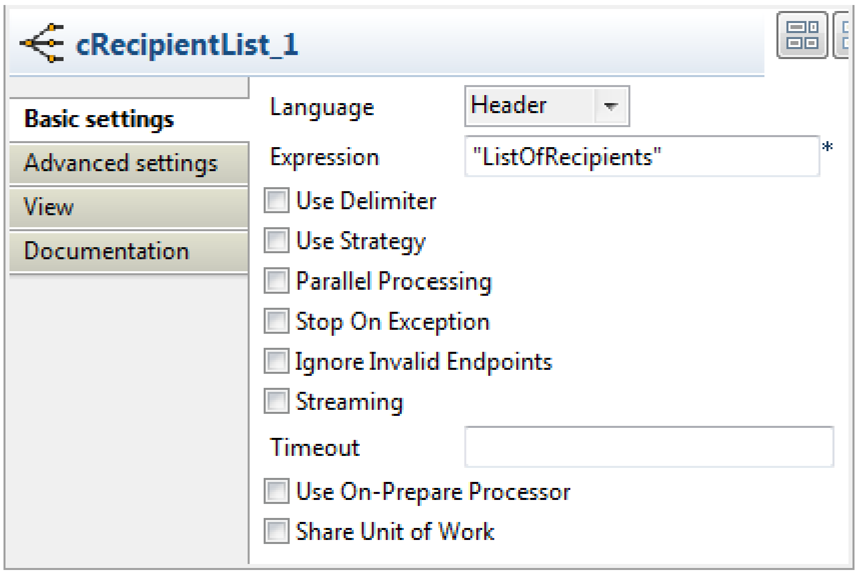
Name	Language	Value
"ListOfRecipients"	Constant	"direct:a,direct:b"

4. Click **[+]** to add a row to the **Headers** table.

   In the **Name** field, enter the header name, for example, *"ListOfRecipients"*.

   In the **Language** list, select *Constant*.

   In the **Value** 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"

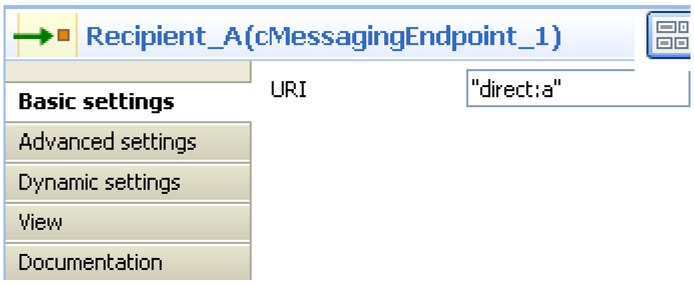
- [ ] Use Delimiter
- [ ] Use Strategy
- [ ] Parallel Processing
- [ ] Stop On Exception
- [ ] Ignore Invalid Endpoints
- [ ] Streaming

Timeout:

- [ ] Use On-Prepare Processor
- [ ] Share Unit of Work

6. In the **Language** list, select **Header**.

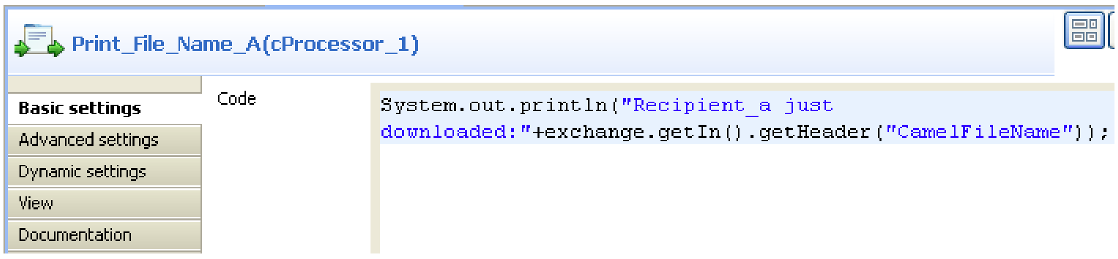
   In the **Expression** field, enter the name of the header that contains the recipients list, that is, *"ListOfRecipients"*.
7. Double-click **Recipient_A** to open its **Basic settings** view in the **Component** tab and define the URI of recipient A.



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.

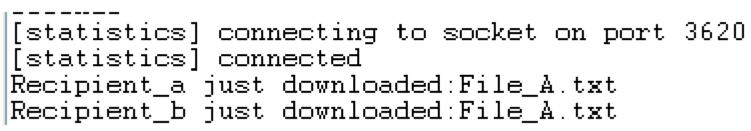
```
public void configure() throws java.lang.Exception {
    from(uriMap.get("Read_Input_cFile_1")).routeId("Read_Input_cFile_1")
            .setHeader("ListOfRecipients").constant("direct:a,direct:b")
            .id("cSetHeader_1")

            .recipientList().header("ListOfRecipients")
            .id("cRecipientList_1");
    from(uriMap.get("Recipient_A_cMessagingEndpoint_1"))
            .routeId("Recipient_A_cMessagingEndpoint_1")
            .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_cMessagingEndpoint_2"))
            .routeId("Recipient_B_cMessagingEndpoint_2")
            .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_2");
}
```

As shown above, the route gets the message from `Read_Input_cFile_1`, and `.setHeader("ListOfRecipients")` 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
```

cRoutingSlip properties

cRoutingSlip is used to route a message or messages consecutively to a series of endpoints.

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.

cRoutingSlip Standard properties

These properties are used to configure cRoutingSlip running in the Standard Job framework.

The Standard cRoutingSlip component belongs to the Routing family.

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

Usage rule	**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.

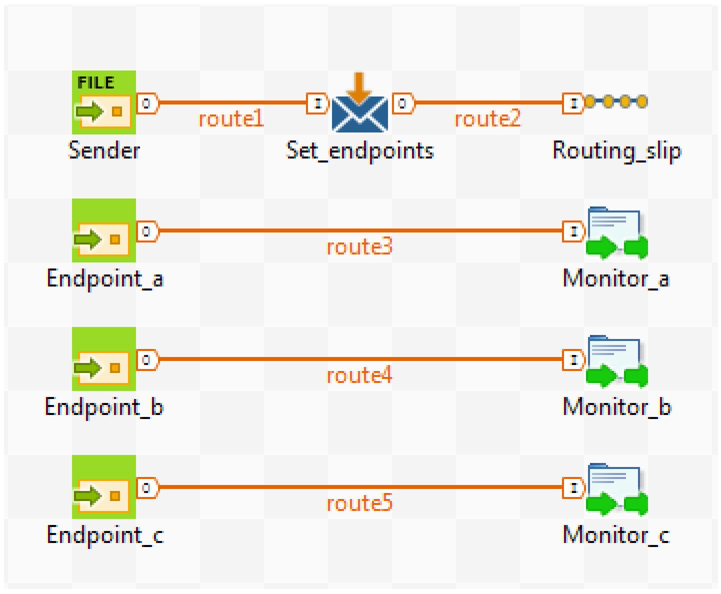
Scenario 1: Routing a message consecutively to a series of endpoints

This scenario applies only to a Talend solution with ESB.

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

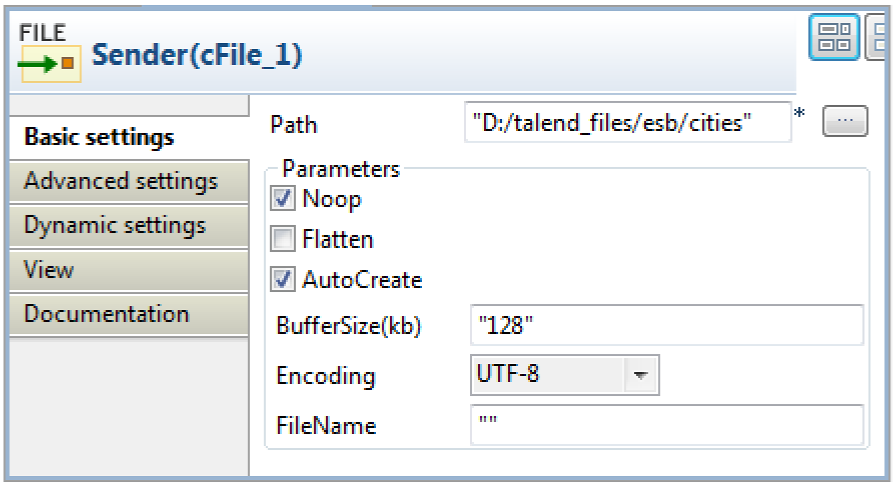
This use case requires a **cFile** component as the message sender, a **cSetHeader** component to define a series of endpoints, a **cRoutingSlip** component to route messages to the endpoints consecutively, three **cMessagingEndpoint** components to retrieve messages routed to the endpoints, and three **cProcessor** components to monitor messages routed to the connected messaging endpoints.



1. From the **Palette**, expand the **Connectivity** 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 **Core** 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 **Custom** 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.



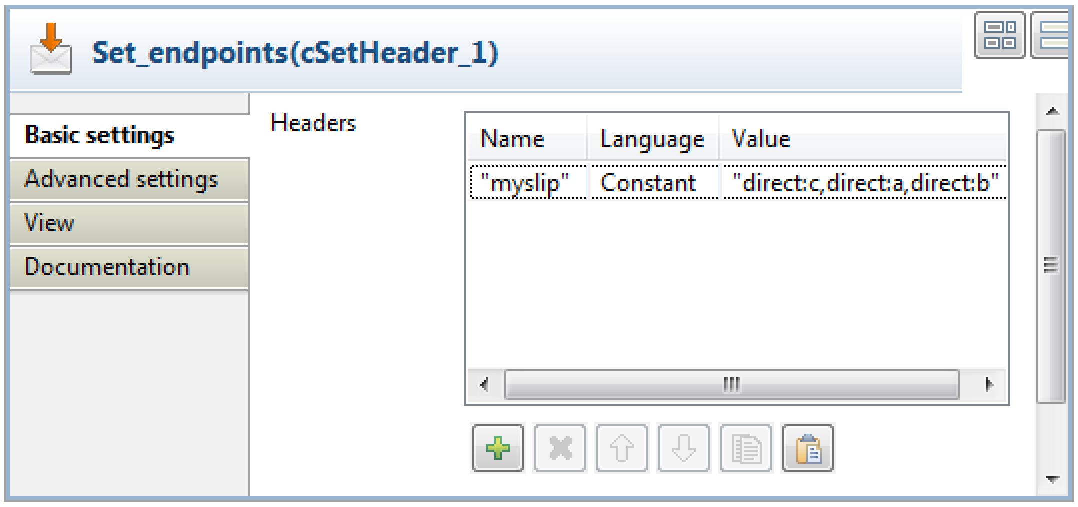
Sender(cFile_1)

Path	"D:/talend_files/esb/cities"
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 *Beijing.xml*, *London.xml*, *Paris.xml*, and *Washington.xml*.

   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.



Set_endpoints(cSetHeader_1)

Headers

Name	Language	Value
"myslip"	Constant	"direct:c,direct:a,direct:b"

4. Click **[+]** to add a row to the **Headers** table.

   In the **Name** field, type in the name of the header you want to add to each message.

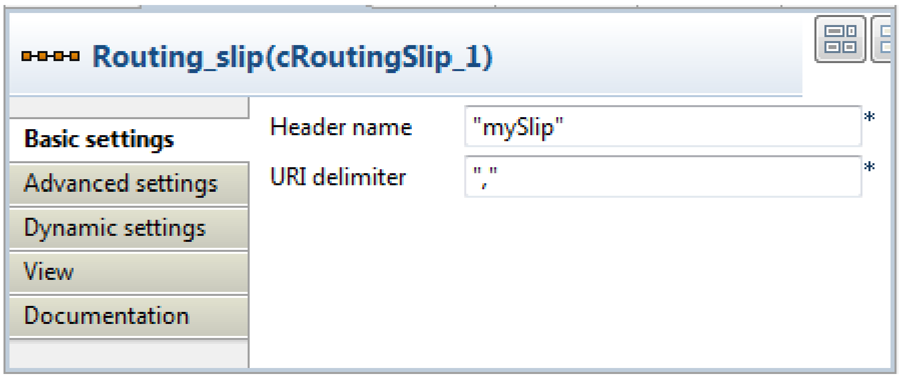
   In this use case, we simply use *mySlip*, which is the default value filled in the **Header name** field of the **cRoutingSlip** component.

5. From the **Language** list box, select the **Constant** or **Simple**, and in the **Value** 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.

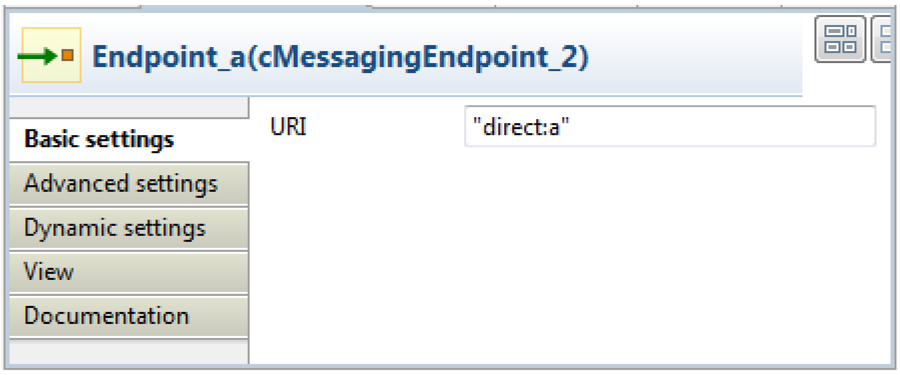

Routing_slip(cRoutingSlip_1)

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

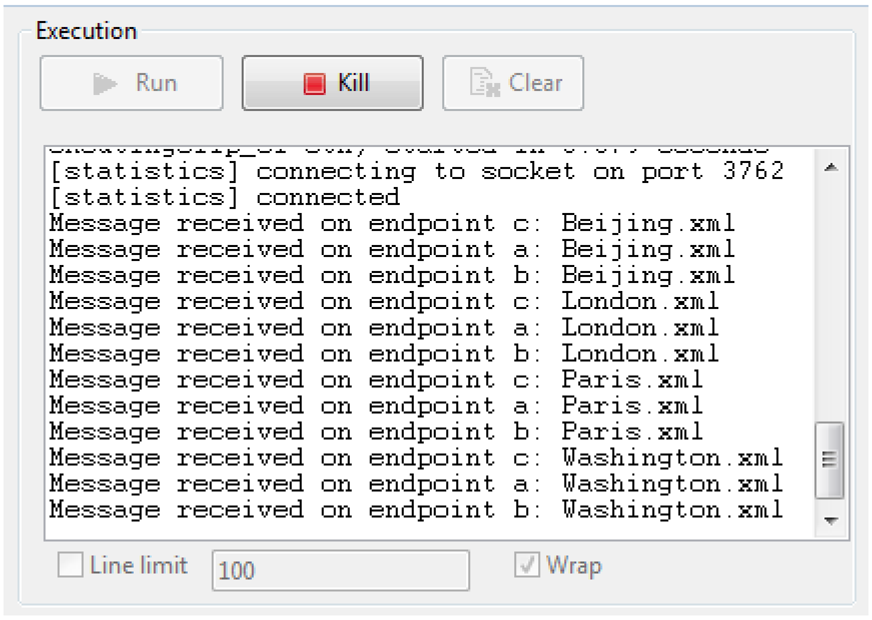
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

```
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

This scenario applies only to a Talend solution with ESB.

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 Studio 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";
```

```
    }
   }
```

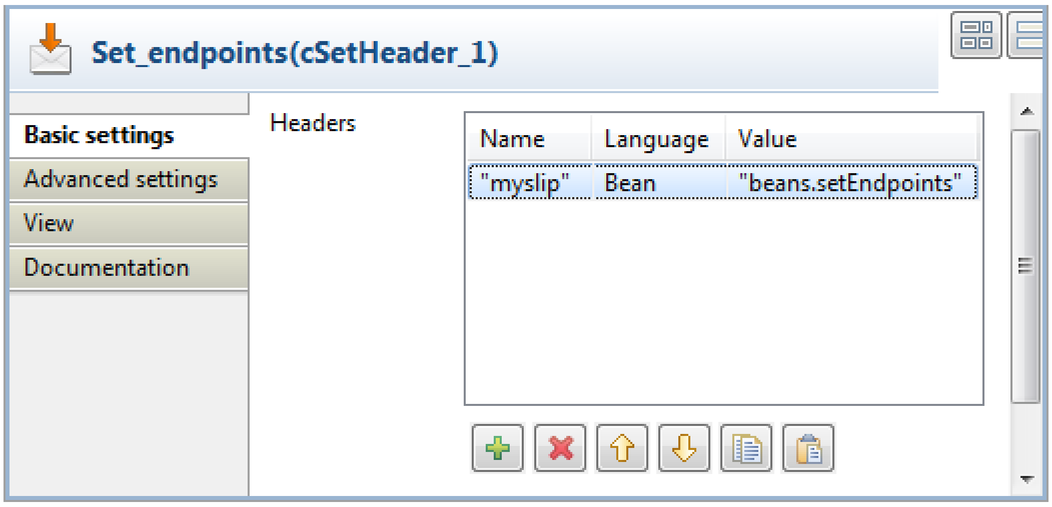
Dropping and linking the components

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)

Headers

Name	Language	Value
"myslip"	Bean	"beans.setEndpoints"

2. Select **Bean** from the **Language** list box, and in the **Value** field, specify the Java Bean that will return the endpoint URIs. In this use case, type in:

   `beans.setEndpoints`
3. Press **Ctrl+S** to save your Route.

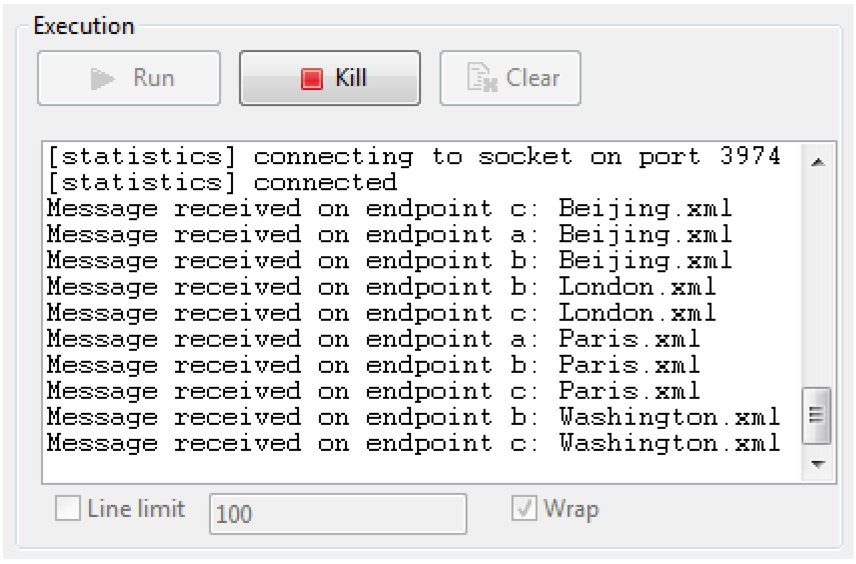
Viewing code and executing the Route

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*.

cSEDA properties

cSEDA allows you to produce and consume messages asynchronously in different threads within a single CamelContext.

cSEDA provides asynchronous SEDA behavior, so that messages are exchanged on a BlockingQueue and consumers are invoked in a separate thread from the producer within a single CamelContext.

cSEDA Standard properties

These properties are used to configure cSEDA running in the Standard Job framework.

The Standard cSEDA component belongs to the Core family.

Basic settings

When using as a start component in a Route:	
Name	Type in any string that uniquely identifies the endpoint.
Specify maximum capacity size	Select this check box to set the maximum number of messages that the SEDA queue can hold. Specify the number in the **Size** field.
Concurrent consumers	Specify the number of concurrent threads processing exchanges.

Wait for task to complete	Specify whether the caller should wait for the asynchronous task to complete or not before continuing. Select from **Always**, **Never** or **IfReplyExpected**. The default option is **IfReplyExpected** which means the caller will only wait if the message is Request-Reply based. For more information about this option, see the site http://camel.apache.org/async.html.
Timeout	Specify the time in milliseconds before a SEDA producer will stop waiting for an asynchronous task to complete. You can disable this option by using `0` or a negative value.
Use multiple consumers	Specifies whether multiple consumers are allowed. If enabled, you can use **cSEDA** for Publish-Subscribe messaging, which means you can send a message to the SEDA queue and have each consumer receive a copy of the message. When enabled, this option should be specified on every consumer endpoint.
Limit concurrent consumers	Whether to limit the number of concurrent consumers to the maximum of 500. By default, an exception will be thrown if a SEDA endpoint is configured with a greater number.
Block when full	Whether a thread that sends messages to a full SEDA queue will block until the queue's capacity is no longer exhausted. By default, an exception will be thrown stating that the queue is full. By enabling this option, the calling thread will block instead and wait until the message can be accepted.
Poll timeout	Specify the timeout in milliseconds used when polling. When a timeout occurs, the consumer can check whether it is allowed to continue running.

	Setting a low value allows the consumer to react more quickly upon shutdown.
When using as a middle or end component in a Route:	
Use Exist cSEDA	Click **[...]** and select the corresponding consumer in the dialog box.

Advanced settings

Arguments	This option is available only when **cSEDA** is used as a start component in the Route. 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/seda.html for available options.

Usage

Usage rule	**cSEDA** is used as a start, middle, or end component in a Route.
Limitation	n/a

Scenario: Using cSEDA, cVM and cDirect to produce and consume messages separately

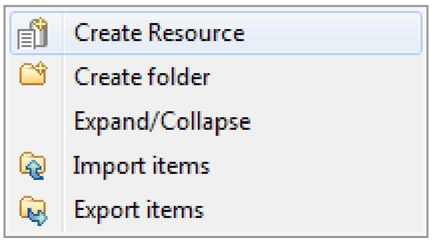
This scenario applies only to a Talend solution with ESB.

In this scenario, we will use a **cTimer** component to trigger a message exchange. The message is routed to a **cSEDA**, a **cVM** and a **cDirect** sequentially with a message body set for each of them, which is then consumed in another thread.

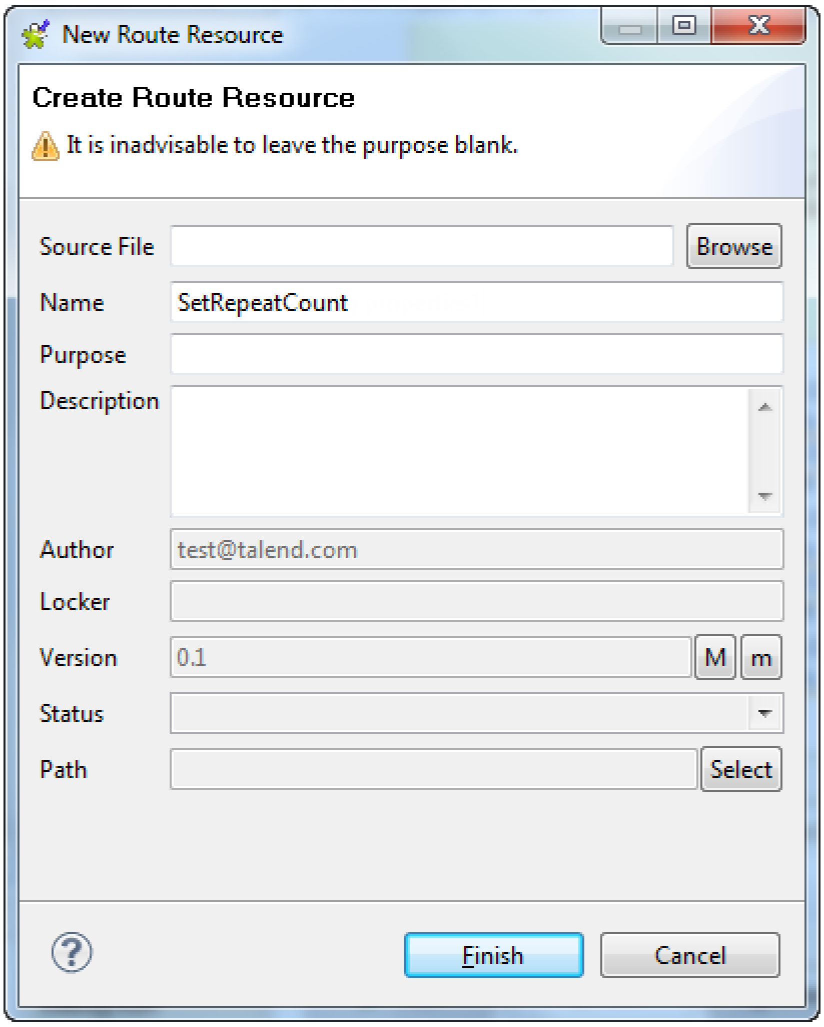
We will create a Route Resource to define the repeat count of the message exchange (the number of times the message should be sent), which will be used by the **cTimer** component.

Creating a Route Resource and calling it in the Route

1. From the repository tree view, right-click the **Resources** node and select **Create Resource** from the context menu.

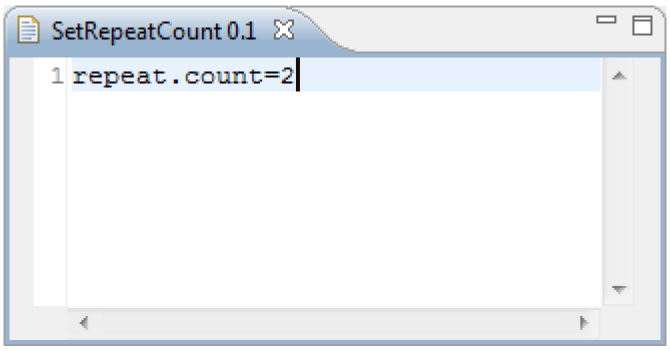


2. The **[New Route Resource]** wizard opens. In the **Name** field, type in a name for the Resource, for example, *SetRepeatCount*. Click **Finish** to close the wizard.



Source File:
Name: SetRepeatCount
Purpose:
Description:
Author: test@talend.com
Locker:
Version: 0.1
Status:
Path:

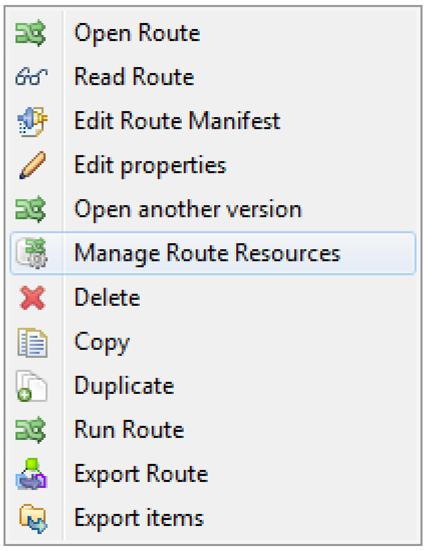
3. Enter `repeat.count=2` in the design workspace to set the repeat count.



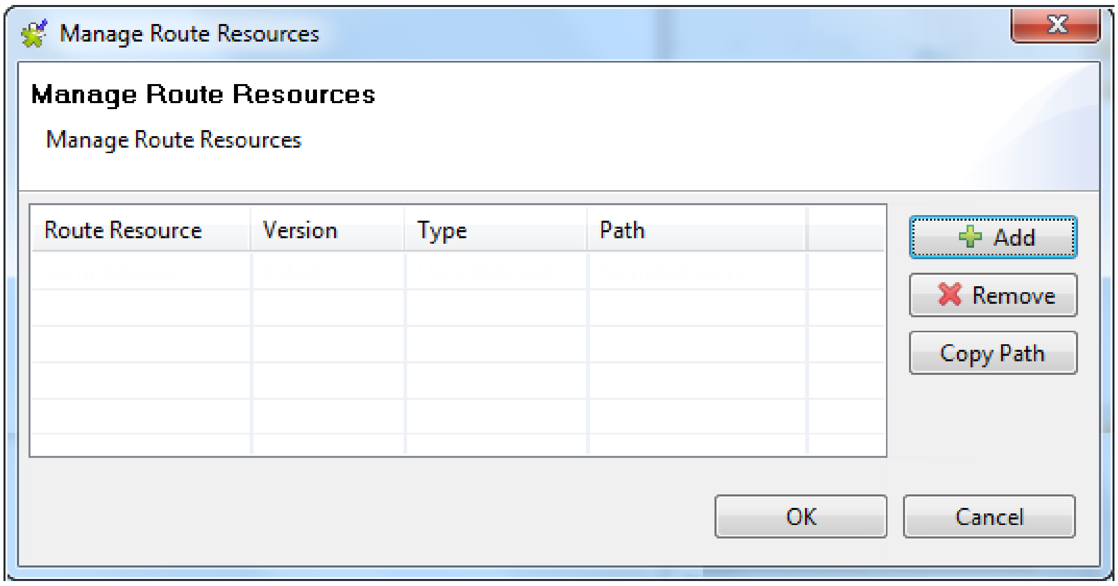
```
repeat.count=2
```

4. Press **Ctrl+S** to save your Route Resource.

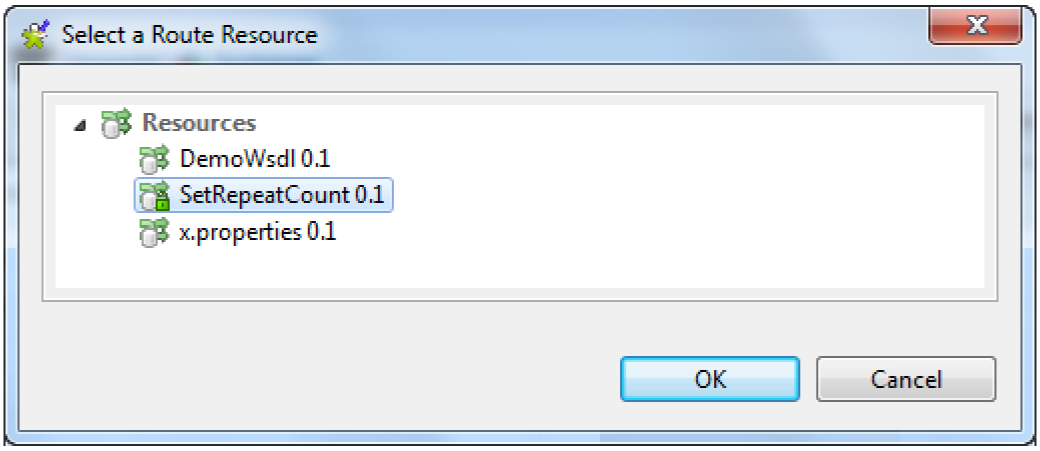
5. Right-click the Route from the repository tree view and select **Manage Route Resources** from the context menu.



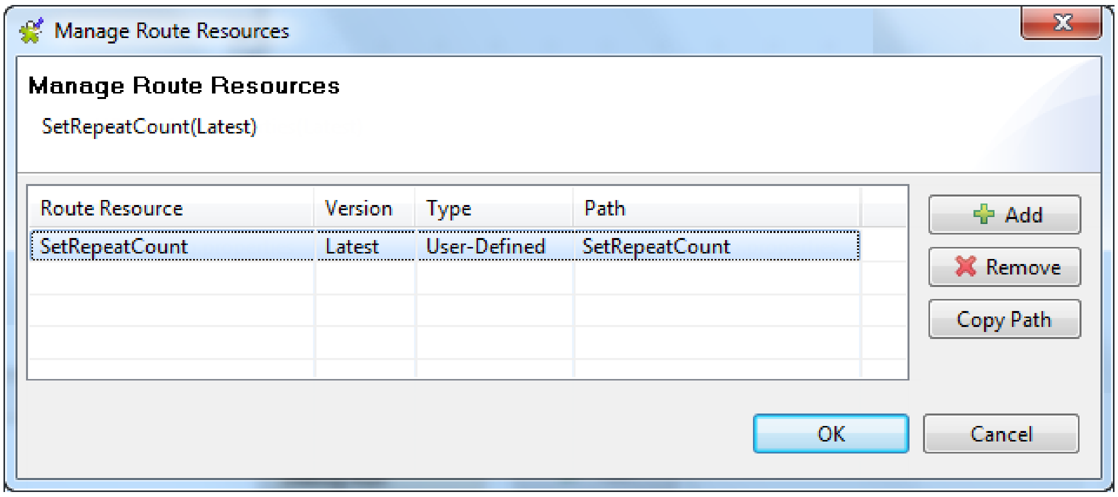
The **[Manage Route Resources]** wizard is opened.



6. Click **Add** and select *SetRepeatCount* from the Resources tree view in the dialog. Click **OK**.



The *SetRepeatCount* Route Resource is added in the table.


Manage Route Resources

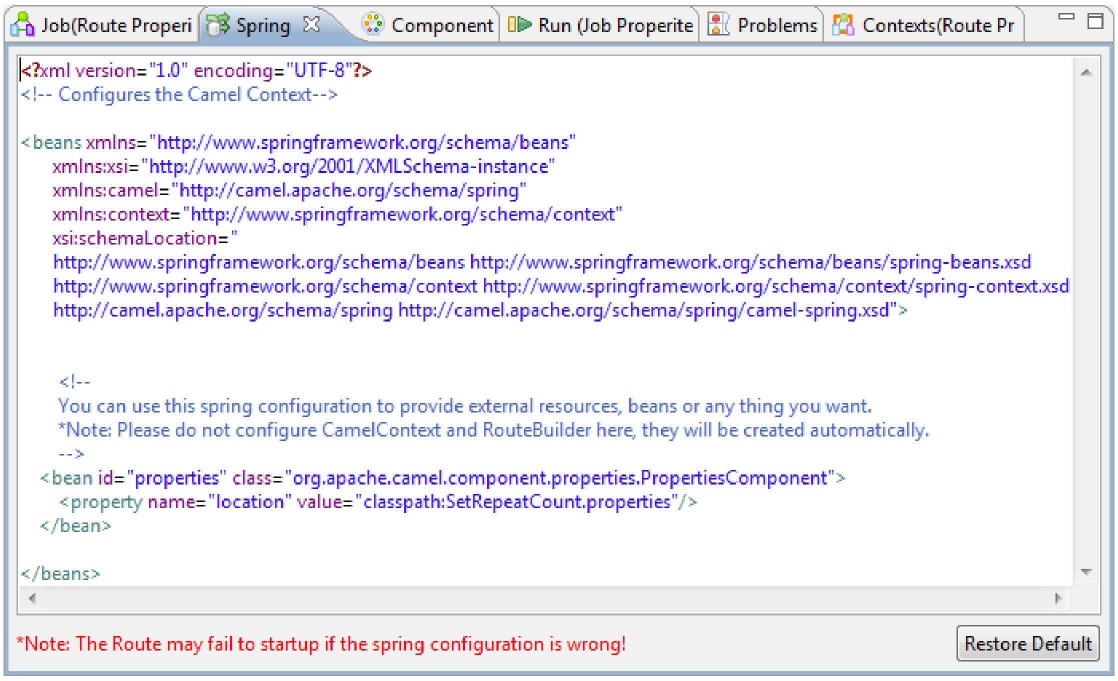
Manage Route Resources

SetRepeatCount(Latest)

Route Resource	Version	Type	Path
SetRepeatCount	Latest	User-Defined	SetRepeatCount

7. Click **OK** to close the wizard.

   For more information about creating and using Route Resources, see *Talend Studio User Guide*.
8. Click the **Spring** tab on the lower half of the design workspace of the Route.



```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Configures the Camel Context-->

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:camel="http://camel.apache.org/schema/spring"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
    http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">


     <!--
     You can use this spring configuration to provide external resources, beans or any thing you want.
     *Note: Please do not configure CamelContext and RouteBuilder here, they will be created automatically.
     -->
  <bean id="properties" class="org.apache.camel.component.properties.PropertiesComponent">
     <property name="location" value="classpath:SetRepeatCount.properties"/>
  </bean>

</beans>
```

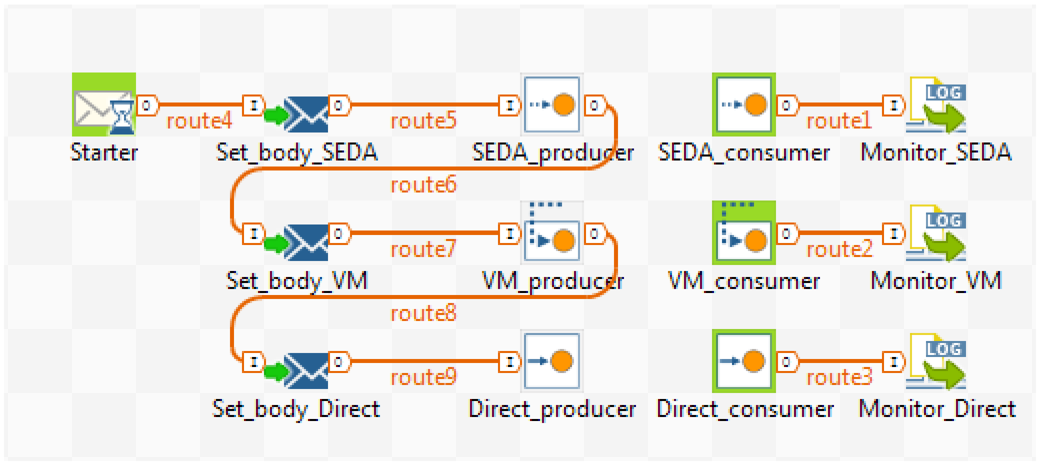
*Note: The Route may fail to startup if the spring configuration is wrong!

9. Enter the following code in this view to call the Route Resource you just created.

```
<bean id="properties"
 class="org.apache.camel.component.properties.PropertiesComponent">
   <property name="location" value="classpath:SetRepeatCount.properties"/>
</bean>
```

For more information about using Spring configuration in a Route, see *Talend Studio User Guide*.

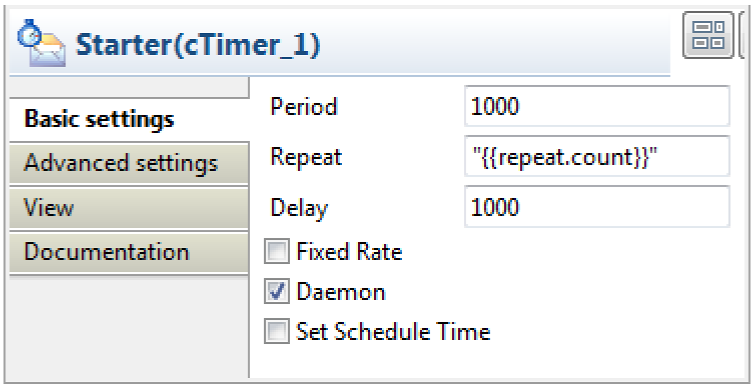
Dropping and linking the components



1. From the **Palette**, drag and drop a **cTimer**, two **cSEDA**, two **cVM**, two **cDirect**, three **cSetbody**, and three **cLog** components onto the design workspace.
2. Link the components using the **Row** > **Route** connection as shown above.
3. Label the components to better identify their roles in the Route.

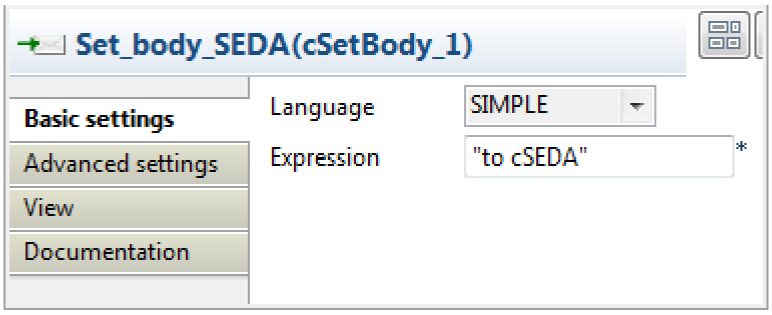
Configuring the components and connections

1. Double-click *Starter* in the design workspace to display its **Basic settings** view in the **Component** tab.



Period	1000
Repeat	"{{repeat.count}}"
Delay	1000
Fixed Rate	☐
Daemon	✓
Set Schedule Time	☐

2. In the **Repeat** field, enter `"{{repeat.count}}"` that is defined in the Route resource.
3. Double-click *Set_body_SEDA* in the design workspace to display its **Basic settings** view in the **Component** tab.



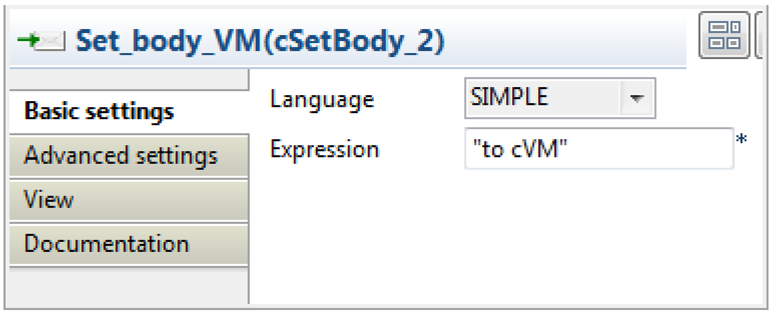
Language	SIMPLE
Expression	"to cSEDA"

4. Select **SIMPLE** in the **Language** list.

   In **Expression** field, enter the `"to cSEDA"` as the message body.

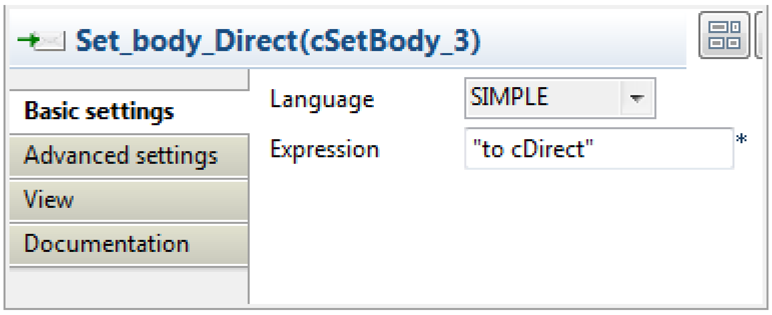
Repeat this step to set the message body in *Set_body_VM* and *Set_body_Direct* as `"to cVM"` and `"to cDirect"` respectively.

Set_body_VM:


Set_body_VM(cSetBody_2)

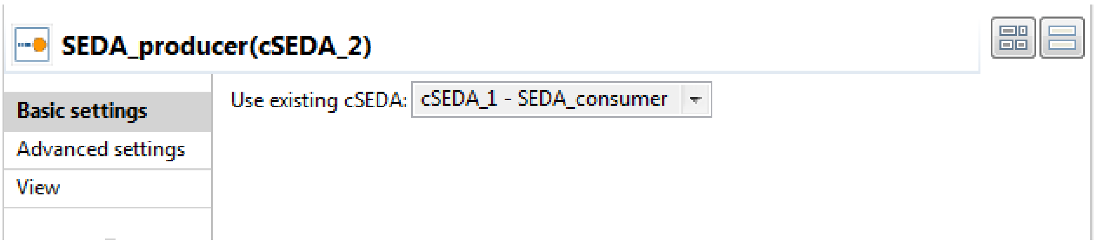
Language: SIMPLE
Expression: "to cVM" *

Set_body_Direct:


Set_body_Direct(cSetBody_3)

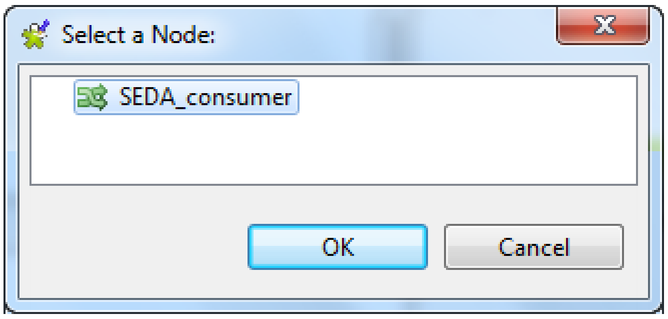
Language: SIMPLE
Expression: "to cDirect" *

5. Double-click *SEDA_producer* to display its **Basic settings** view in the **Component** tab.

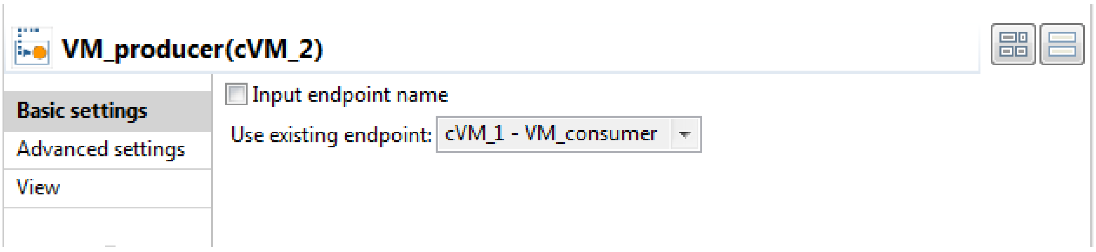

SEDA_producer(cSEDA_2)

Use existing cSEDA: cSEDA_1 - SEDA_consumer

6. Click **[...]** and select *SEDA_consumer* in the **[Select a Node:]** wizard, which will consume the message that is sent to *SEDA_producer*.



Repeat this step to select the node *VM_consumer* for *VM_producer*.

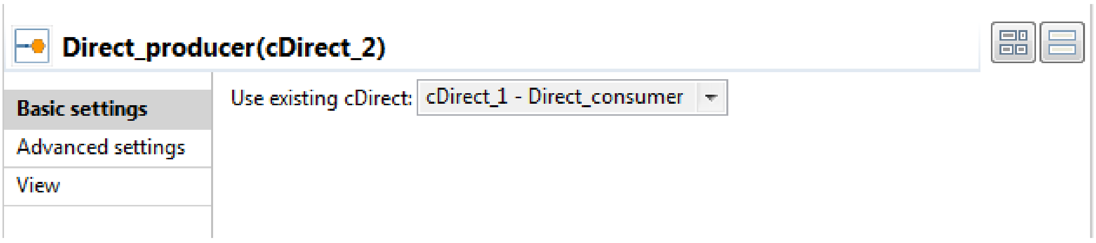

VM_producer(cVM_2)

Basic settings | Advanced settings | View

☐ Input endpoint name

Use existing endpoint: cVM_1 - VM_consumer

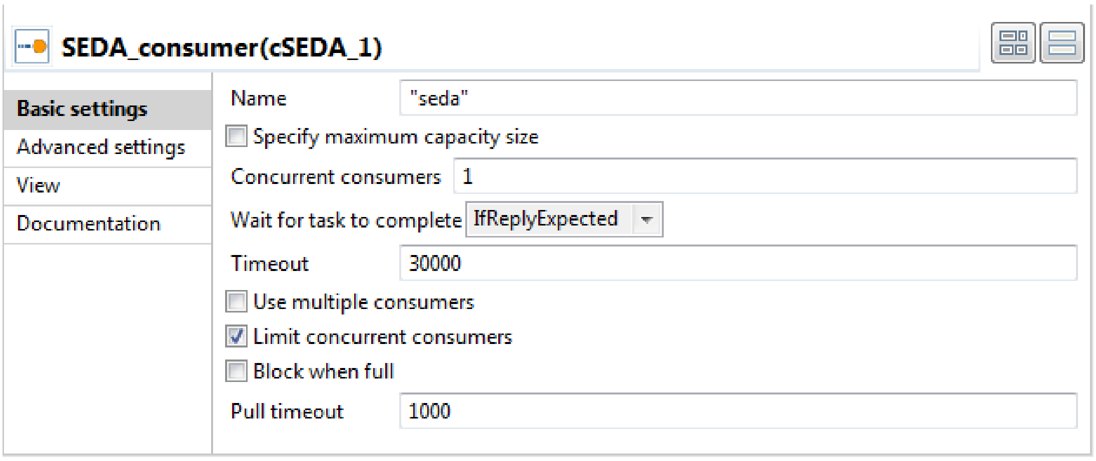
Select the node *Direct_consumer* for *VM_producer*.


Direct_producer(cDirect_2)

Basic settings | Advanced settings | View

Use existing cDirect: cDirect_1 - Direct_consumer

7. Double-click *SEDA_consumer* to display its **Basic settings** view in the **Component** tab.


SEDA_consumer(cSEDA_1)

Basic settings | Advanced settings | View | Documentation

Name: "seda"

☐ Specify maximum capacity size

Concurrent consumers: 1

Wait for task to complete: IfReplyExpected

Timeout: 30000

☐ Use multiple consumers

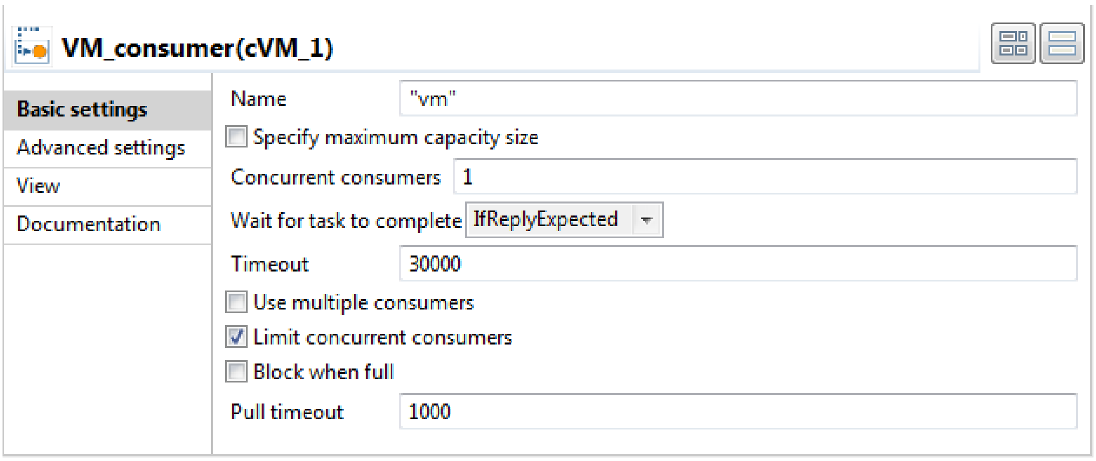
☑ Limit concurrent consumers

☐ Block when full

Pull timeout: 1000

8. In the **Name** field, type in `"seda"` to identify this endpoint. Keep the default settings of the other options.

   Repeat this step to give the name `"vm"` to *VM_consumer*.


VM_consumer(cVM_1)

Basic settings | Advanced settings | View | Documentation

Name: "vm"

☐ Specify maximum capacity size

Concurrent consumers: 1

Wait for task to complete: IfReplyExpected

Timeout: 30000

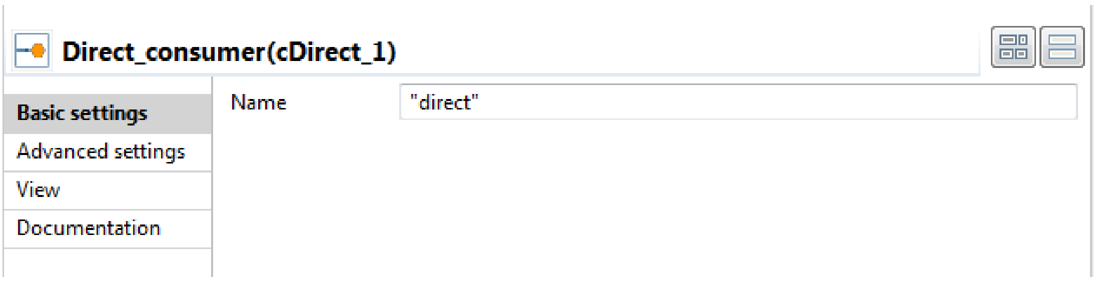
☐ Use multiple consumers

☑ Limit concurrent consumers

☐ Block when full

Pull timeout: 1000

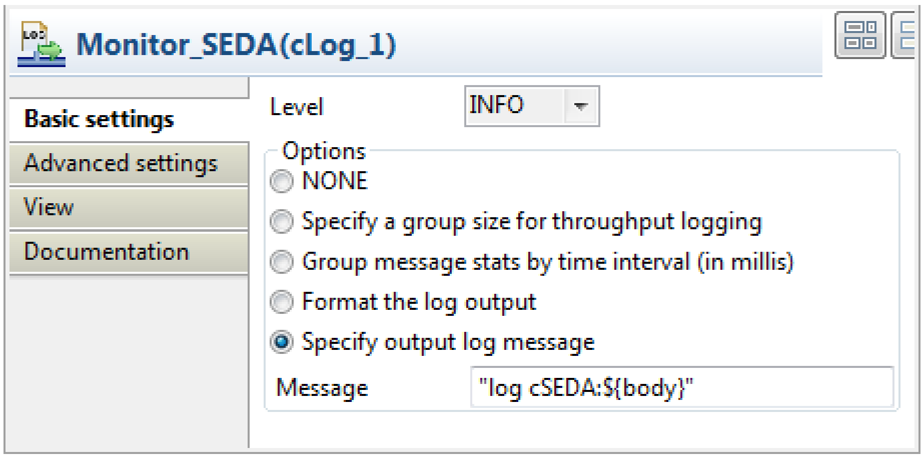
Give the name `"direct"` to *Direct_consumer*.



Direct_consumer(cDirect_1)

Name "direct"

9. Double-click *Monitor_SEDA* to display its **Basic settings** view in the **Component** tab.



Monitor_SEDA(cLog_1)

Level INFO

Options
- ( ) NONE
- ( ) Specify a group size for throughput logging
- ( ) Group message stats by time interval (in millis)
- ( ) Format the log output
- [✓ Specify output log message]

Message "log cSEDA:${body}"

10. Select **INFO** in the **Level** list and select **Specify output log message**. In the **Message** field, enter `"log cSEDA:${body}"` to print the message body in the console.

    Repeat this step to specify the output message for *Monitor_VM* and *Monitor_Direct* as `"log cVM:${body}"` and `"log cDirect:${body}"` respectively.

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

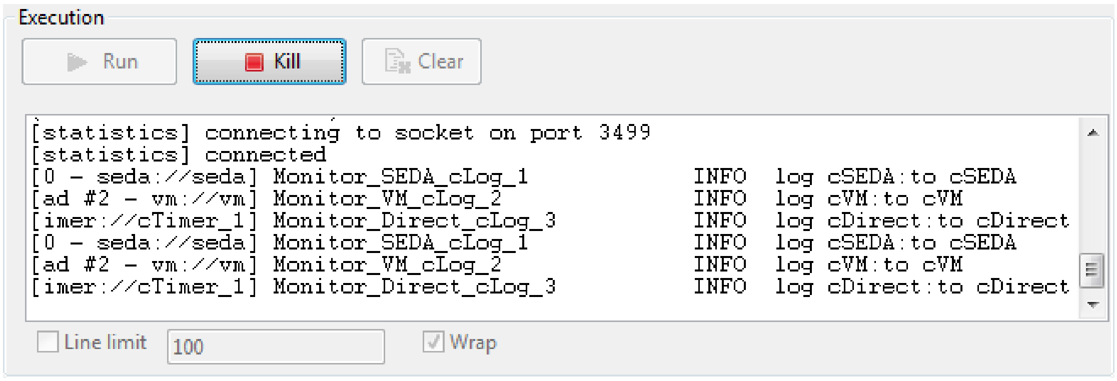
Viewing code and executing the Route

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

```
public void configure() throws java.lang.Exception {
    from(uriMap.get("Direct_consumer_cDirect_1"))
            .routeId("Direct_consumer_cDirect_1")
            .log(org.apache.camel.LoggingLevel.INFO,
                    "Monitor_Direct_cLog_3", "log cDirect:${body}")

            .id("cLog_3");
    from(uriMap.get("SEDA_consumer_cSEDA_1"))
            .routeId("SEDA_consumer_cSEDA_1")
            .log(org.apache.camel.LoggingLevel.INFO, "Monitor_SEDA_cLog_1",
                    "log cSEDA:${body}")

            .id("cLog_1");
    from(uriMap.get("Starter_cTimer_1")).routeId("Starter_cTimer_1")
            .setBody().simple("to cSEDA").id("cSetBody_1")
            .to(uriMap.get("SEDA_consumer_cSEDA_1")).id("cSEDA_2").setBody()
            .simple("to cVM").id("cSetBody_2").to(uriMap.get("VM_consumer_cVM_1"))
            .id("cVM_2").setBody().simple("to cDirect").id("cSetBody_3")
            .to(uriMap.get("Direct_consumer_cDirect_1")).id("cDirect_2");
    from(uriMap.get("VM_consumer_cVM_1"))
            .routeId("VM_consumer_cVM_1")
            .log(org.apache.camel.LoggingLevel.INFO, "Monitor_VM_cLog_2",
                    "log cVM:${body}")

            .id("cLog_2");
}
```

As shown in the code, a message route is built `from` `"Starter_cTimer_1"`, set the message body `"to cSEDA"` by `"cSetBody_1"`, and sent to `cSEDA_2`, which is mapped to `"SEDA_consumer_cSEDA_1"`. The message is then sent to `cVM_2`, `cDirect_2` sequentially which is mapped to its corresponding consumer with a new message body. On the consumer side, the message body `from` each consumer is logged by the corresponding monitor.

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 3499
[statistics] connected
[0 - seda://seda] Monitor_SEDA_cLog_1          INFO  log cSEDA:to cSEDA
[ad #2 - vm://vm] Monitor_VM_cLog_2            INFO  log cVM:to cVM
[imer://cTimer_1] Monitor_Direct_cLog_3        INFO  log cDirect:to cDirect
[0 - seda://seda] Monitor_SEDA_cLog_1          INFO  log cSEDA:to cSEDA
[ad #2 - vm://vm] Monitor_VM_cLog_2            INFO  log cVM:to cVM
[imer://cTimer_1] Monitor_Direct_cLog_3        INFO  log cDirect:to cDirect
```

RESULT: The message that is sent to *SEDA_producer*, *VM_producer*, and *Direct_producer* is consumed by *SEDA_consumer*, *VM_consumer*, and *Direct_consumer* respectively. The message exchange is triggered twice as set in the Route Resource *SetRepeatCount*.

cSetBody properties

cSetBody is used to set the message body in the Route.

cSetBody replaces the payload of each message sent to it.

cSetBody Standard properties

These properties are used to configure cSetBody running in the Standard Job framework.

The Standard cSetBody component belongs to the Core family.

Basic settings

Language	Select the language of the expression you use to set the content for matched messages, from **None**, **Bean**, **CONSTANT**, **CorrelationID**, **EL**, **GROOVY**, **HEADER**, **JAVASCRIPT**, **JoSQL**, **JSonPath**, **JXPATH**, **MVEL**, **OGNL**, **PHP**, **PROPERTY**, **PYTHON**, **RUBY**, **SIMPLE**, **SpEL**, **SQL**, **XPATH**, and **XQUERY**.
Expression	Type in the expression to set the message content.
Correlation expression/Add Namespaces	This option appears when **XPath** is selected in the **Language** list. Select this check box to add namespaces for the Xpath expression. Click **[+]** to add as many namespaces as required to the table and define the prefix and URI in the corresponding columns.

Usage

Usage rule	**cSetBody** is used as a middle component in a Route.
Limitation	n/a

Scenario: Replacing the content of messages with their extracts

This scenario applies only to a Talend solution with ESB.

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.

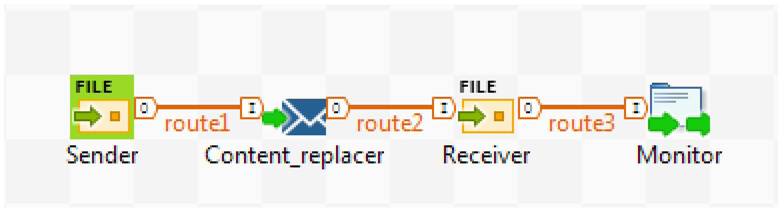
The following is an example of the XML files used in this use case:

```
<people>
  <person>
    <firstName>Pierre</firstName>
    <lastName>Dubois</lastName>
    <city>Paris</city>
```

```
        </person>
    </people>
```

Dropping and linking the components

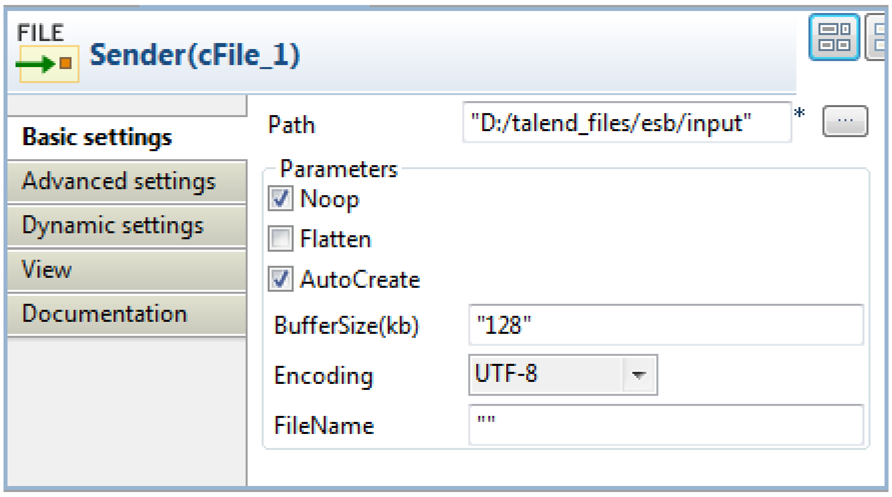
This use case uses two **cFile** components, one as the message sender and the other as the receiver, a **cSetBody** component to replace the content of the messages on route, and a **cProcessor** component to display the new content of the messages routed to the receiving endpoint.



1. From the **Palette**, expand the **Connectivity** folder, and drop two **cFile** components onto the design workspace.
2. From the **Core** folder, drop a **cSetBody** component onto the design workspace, between the two **cFile** components.
3. From the **Custom** 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

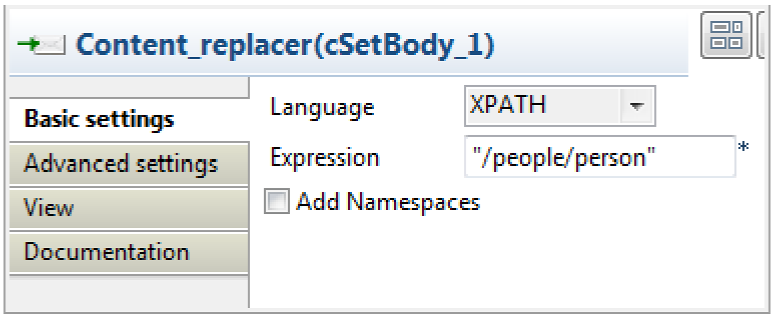
1. Double-click the **cFile** component labeled *Sender* to display its **Basic settings** view in the **Component** tab.



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



Content_replacer(cSetBody_1)

Language: XPATH

Expression: "/people/person"

Add Namespaces

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:

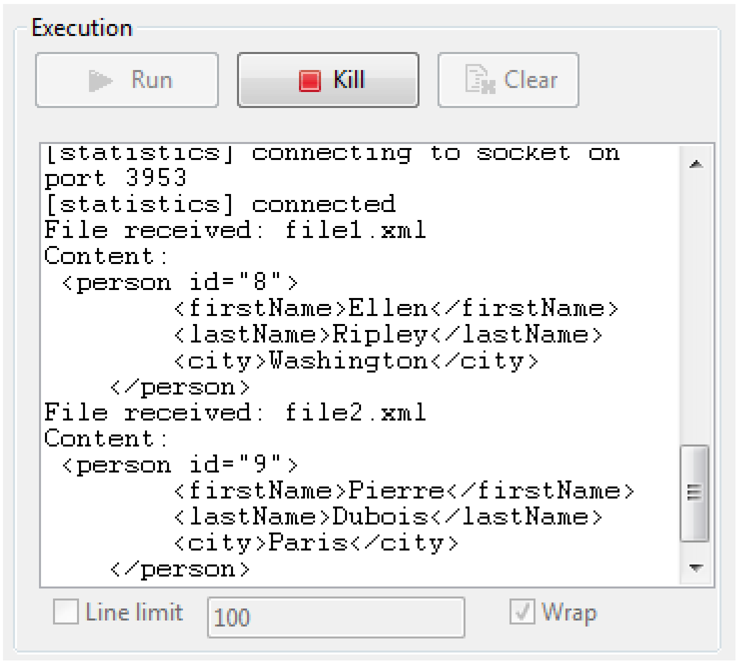
   ```
   System.out.println("File received: " +

   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 properties

cSetHeader is used to set headers or customize the default headers, if any, on each message sent to it for subsequent message processing.

cSetHeader sets headers on each message sent to it.

cSetHeader Standard properties

These properties are used to configure cSetHeader running in the Standard Job framework.

The Standard cSetHeader component belongs to the Core family.

Basic settings

Headers	Click **[+]** to add as many headers as required to the table.
	Name: Type in a name for the message header. The header name *CorrelationID* is reserved. The value of this header will be overridden by the correlation ID of the message if it exist.
	Language: Select the language of the expression you use from **None**, **Bean**, **Constant**, **CorrelationID**, **EL**, **Groovy**, **Header**,

	JavaScript, **JoSQL**, **JSonPath**, **JXPath**, **MVEL**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**. Select **Bean** if you want to call a predefined Java Bean to return the header value. Select **CorrelationID** to use the existing correlation ID of the message as the header value if the correlation ID is available in the closest **cSOAP** connected to this component. For more information about the **cSOAP** component, see cSOAP properties on page 53.
	Value: Type in the expression to set the value of the message header, or the Bean class that will return a value for the message header, in the form of *beans.BEAN_NAME*. If **CorrelationID** is selected in the **Language** list, this field is grayed out. The existing correlation ID from the closest **cSOAP** connected to this component will be set as the header value. For more information about the **cSOAP** component, see cSOAP properties on page 53.
Add Namespaces	Select this check box to add namespaces for the expression. Click **[+]** to add as many namespaces as required to the table and define the prefix and URI in the corresponding columns.

Usage

Usage rule	**cSetHeader** is used as a middle component in a Route.
Limitation	n/a

Related scenarios

For more scenarios, see:

Scenario: Using cTalendJob to call a DI Job

Scenario 1: Routing a message consecutively to a series of endpoints on page 177

Scenario 2: Routing each message conditionally to a series of endpoints on page 181

cSetProperty properties

cSetProperty is used to set properties for each message sent to it for subsequent message processing.

cSetProperty sets properties for each message sent to it.

cSetProperty Standard properties

These properties are used to configure cSetProperty running in the Standard Job framework.

The Standard cSetProperty component belongs to the Core family.

Basic settings

Properties	Click **[+]** to add as many properties as required to the table:
	Name: define the name of the property.
	Language: select the language of the expression you use to set the value for property, from **None**, **Bean**, **Constant**, **ESB[CorrelationID]**, **EL**, **Groovy**, **Header**, **JavaScript**, **JoSQL**, **JSonPath**, **JXPath**, **MVEL**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**.
	Value: enter the value of the property.
Add Namespaces	This option appears when **XPath** is selected in the **Language** list. Select this check box to add namespaces for the properties. Click **[+]** to add as many namespaces as required to the table and define the prefix and URI in the corresponding columns.

Usage

Usage rule	**cSetProperty** is used as a middle component in a Route.
Limitation	n/a

Related scenarios

No scenario is available for the Standard version of this component yet.

cSplitter properties

cSplitter separates multiple elements of a message so that they can be handled and treated differently in individual routes.

cSplitter splits a message into several sub-messages according to a condition.

cSplitter Standard properties

These properties are used to configure cSplitter running in the Standard Job framework.

The Standard cSplitter component belongs to the Routing family.

Basic settings

Language	Select the language of the expression you want to use to split your messages, from **None**, **Constant**, **EL**, **Groovy**, **Header**, **JavaScript**, **JoSQL**, **JSonPath**, **JXPath**, **MVEL**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**.
Correlation expression/Expression	Type in the expression to use to split the messages.
Correlation expression/Use Result Class Type	This option appears when **XPath** is selected in the **Language** list. Select this check box to set the result type of the sub-messages in the field that appears. The default native XML objects `org.w3c.dom.NodeList` will be used if not specified.

Correlation expression/Add Namespaces	This option appears when **XPath** is selected in the **Language** list. Select this check box to add namespaces for the Xpath expression. Click **[+]** to add as many namespaces as required to the table and define the prefix and URI in the corresponding columns.
Use Strategy	Select this check box to refer to an aggregation strategy to assemble the replies from the sub-messages into a single outgoing message from the splitter. Enter the ID of the aggregation strategy in the **Strategy** field. The sub-message replies will be aggregated in the order they come back if **Streaming** is enabled. If not, the sub-message replies will be aggregated in the same order as they were split.
Parameters/Parallel Processing	Select this check box to process the sub-messages concurrently. The caller thread will wait until all sub-messages have been fully processed before it continues.
Parameters/Stop on Exception	Select this check box to stop processing immediately when an exception occurs.
Parameters/Streaming	Select this check box to split the message in a streaming fashion, which means it will split the input message in chunks. It is recommended to enable this option when processing big messages.
Parameters/Share Unit of Work	Select this check box to share the unit of work between the parent exchange and each split exchange. For more information and an use case of this option, see the site http://camel.apache.org/splitter.html.
Parameters/Timeout	Specify a total timeout in millisecond. If the message is not split and processed within the

	given time frame, the timeout triggers and the splitter breaks out.

Usage

Usage rule	**cSplitter** is used as a middle component in a Route.
Connections	**split**
Route	Select this link to route all the messages from the sender to the next endpoint.
Limitation	n/a

cStop properties

cStop is used to stop a message routing.

cStop stops the Route to which it is connected.

cStop Standard properties

These properties are used to configure cStop running in the Standard Job framework.

The Standard cStop component belongs to the Miscellaneous family.

Usage

Usage rule	**cStop** is not a start component, but it can be a middle or end component in a Route.

Related scenario:

For a related scenario, see Scenario: Intercepting several routes and redirect them in a single new route of cIntercept properties on page 97.

cTalendJob properties

cTalendJob allows you to exchange messages between a Data Integration Job and a Mediation Route.

cTalendJob calls a Data Integration Job either from the repository or exported as an OSGI Bundle For ESB. For more information on how to build a Job and how to export a Job as an OSGI Bundle for ESB, see *Talend Studio User Guide*.

cTalendJob Standard properties

These properties are used to configure cTalendJob running in the Standard Job framework.

The Standard cTalendJob component belongs to the Talend family.

Basic settings

Repository	Select this option to call a Job from the Repository.
External	Select this option to call a Job that is exported as an OSGI Bundle For ESB.
Repository/Use Selected Context	This field appears when **Repository** is selected. Select this check box to use the context that is selected in the **Context** list when executing the Job.
Repository/Use Route Context	This field appears when **Repository** is selected. Select this check box to use the Job context that has the same name as the one that is used in the Route when executing the Job. **Note:** If context does not exist in the Job, null values of the context parameters will be used during the Job execution. Make sure that you have the needed context in the Job.
Repository/Use Job Context	This field appears when **Repository** is selected. Select this check box to use the selected context on the Job side when executing the Job.
Repository/Job	This field appears when **Repository** is selected. Click **[...]** to show the **[Assign Job]** wizard. Choose between **Create a new Job and Assign it to this cTalendJob component** and **Assign an existing Job to this cTalendJob component** and follow the prompts. **Warning:** When assigning an existing Job to **cTalendJob**, only the Jobs with the **tRouteInput** component can be selected.

	You can double click **cTalendJob** to open the referenced Job, or right-click **cTalendJob** and select **Open Job in Integration** in the context menu open it.
Repository/Version	This field appears when **Repository** is selected. Select the version of the Job if more than one version of the Job is available.
Repository/Context	This field appears when **Repository** is selected. Select from the list the context to use to execute the referenced Job. ⚠ **Warning:** This option works only when **Use Selected Context** is selected.
External Jar/Library	This field appears when **External** is selected. Select the library you want to import from the list, or click on the **[...]** button to import the jar library of your Job.
External Jar/Job	This field appears when **External** is selected. 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"/>`.
External Jar/Context	This field appears when **External** is selected. Type in the name of the context to use to execute the referenced Job.
Context Param	Use this table to change the variable values of the specified context in the referenced Job.

	Click **[+]** to add as many rows as required to the table. Select the context variable that you want to change in the **Parameters** list of each row, and enter the value you want to give it in the **Values** field. This value will replace the one that is defined on the Job side.

Advanced settings

Propagate Header	Select this check box to pass the message header to the referenced Job as a context variable.
Fast Job Invocation	Select this check box to bind the life cycle of the embedded Talend Job to the start and stop state changes of the connected Talend Camel Endpoint. When the Route is started, the endpoint for the embedded Job is also started, and the Job instance is created and ready for receiving message exchanges. With this option enabled, while the Route is active, the embedded Job keeps long-living resources and refreshes short-living resources between invocations. In this case the database access objects are only kept connected while data are sent or received, which avoids overhead and performance loss with Jobs containing database assess resources that are expensive on creation. **Warning:** Due to the variety of possible Jobs, there is no warranty that a specific Job will work as expected with the **Fast Job Invocation** option activated. Therefore, Jobs using this option need to be tested well for proper execution, and in failure case this option needs to be de-activated.

	⚠ **Warning:** In combination with the **Fast Job Invocation** option, the **Propagate Header** option may not work as expected. The combination needs to be tested well for the specific Job, and in failure case the **Fast Job Invocation** option needs to be de-activated.

Usage

Usage rule	**cTalendJob** can be a start, middle or end component in a Route. It is mandatory that a **tRouteInput** component is used in the Data Integration Job. The reason for it is that this will prevent the referenced Job from starting automatically when deployed in Talend Runtime. Instead it will only start when it gets called by the Route.
Limitation	n/a

cThrottler properties

cThrottler allows you to limit the number of messages flowing to a specific endpoint in order to prevent it from getting overloaded.

cThrottler is designed to limit the number of messages flowing to the subsequent endpoint.

cThrottler Standard properties

These properties are used to configure cThrottler running in the Standard Job framework.

The Standard cThrottler component belongs to the Routing family.

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

Usage rule	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**
Route	Select this link to route all the messages from the sender to the next endpoint.
Limitation	n/a

cTimer properties

The **cTimer** component allows you to schedule message exchanges.

The **cTimer** component generates message exchanges when a timer triggers.

cTimer Standard properties

These properties are used to configure cTimer running in the Standard Job framework.

The Standard cTimer component belongs to the Orchestration family.

Basic settings

Period	Fill this field with an integer (in milliseconds) to generate message exchanges every period.
Repeat	Specifies a maximum limit of message exchange numbers. A value of zero or negative will generate message exchanges forever.
Delay	The number of milliseconds to wait before the first message exchange is generated. This option

	should not be used with the **Set Schedule Time** option.
Fixed Rate	Select this check box to generate message exchanges at regular intervals, separated by the specified period.
Daemon	Specify whether the thread associated with the timer endpoint runs as a daemon.
Set Schedule Time	Select this check box to specify the time that the first message exchange should be generated. In the **Time** field, enter the time using the pattern `yyyy-MM-dd HH:mm:ss` or `yyyy-MM-dd'T'HH:mm:ss`.

Usage

Usage rule	**cTimer** can only be used as a start component in a Route.
Limitation	n/a

Related Scenario:

For a related scenario, see Scenario: Using cDataset to create messages.

cTry properties

cTry is designed to build Try/Catch/Finally blocks to handle exceptions.

cTry offers the Java equivalent exception handling abilities by building Try/Catch/Finally blocks to isolate the part of your Route likely to generate an error, catch the errors, and execute final instructions regardless of the errors.

cTry Standard properties

These properties are used to configure cTry running in the Standard Job framework.

The Standard cTry component belongs to the Exception Handling family.

Usage

Usage rule	**cTry** is used as a middle component in a Route.
Connections	**Try**

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. **Note:** 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. **Note:** 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

cVM properties

cVM allows you to produce and consume messages asynchronously in different threads across CamelContext. You can use this mechanism to communicate across Web applications.

cVM provides asynchronous SEDA behavior, so that messages are exchanged on a BlockingQueue and consumers are invoked in a separate thread from the producer across CamelContext instances.

cVM Standard properties

These properties are used to configure cVM running in the Standard Job framework.

The Standard cVM component belongs to the Core family.

Basic settings

When using as a start component in a Route:	
Name	Type in any string that uniquely identifies the endpoint.

Specify maximum capacity size	Select this check box to set the maximum number of messages that the SEDA queue can hold. Specify the number in the **Size** field.
Concurrent consumers	Specify the number of concurrent threads processing exchanges.
Wait for task to complete	Specify whether the caller should wait for the asynchronous task to complete or not before continuing. Select from **Always**, **Never** or **IfReplyExpected**. The default option is **IfReplyExpected** which means the caller will only wait if the message is Request-Reply based. For more information about this option, see the site http://camel.apache.org/async.html.
Timeout	Specify the time in milliseconds before a SEDA producer will stop waiting for an asynchronous task to complete. You can disable this option by using `0` or a negative value.
Use multiple consumers	Specifies whether multiple consumers are allowed. If enabled, you can use **cVM** for Publish-Subscribe messaging, which means you can send a message to the SEDA queue and have each consumer receive a copy of the message. When enabled, this option should be specified on every consumer endpoint.
Limit concurrent consumers	Whether to limit the number of concurrent consumers to the maximum of 500. By default, an exception will be thrown if a SEDA endpoint is configured with a greater number.
Block when full	Whether a thread that sends messages to a full SEDA queue will block until the queue's capacity is no longer exhausted. By default, an exception will be thrown stating that the queue is full. By enabling this option, the calling thread will

	block instead and wait until the message can be accepted.
Poll timeout	Specify the timeout in milliseconds used when polling. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.
When using as a middle or end component in a Route:	
Input endpoint name	Select this check box to enter the name of the corresponding consumer in the **Name** field.
Use Exist cVM	This option appears when the **Input endpoint name** check box is cleared. Click **[...]** and select the corresponding consumer in the dialog box.
Name	This option appears when the **Input endpoint name** check box is selected. Enter the name of the consumer in this field directly.

Advanced settings

Arguments	This option is available only when **cVM** is used as a start component in the Route. 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/vm.html for available options.

Usage

Usage rule	**cVM** is used as a start, middle, or end component in a Route.
Limitation	n/a

Related scenario:

For a related scenario, see Scenario: Using cSEDA, cVM and cDirect to produce and consume messages separately on page 186.

cWireTap properties

cWireTap is used to route messages to a separate endpoint while forwarded to the ultimate destination.

cWireTap wiretaps messages to a user defined URI while they are sent to their original endpoint. It also allows you to populate a new message to this wiretap URI concurrently.

cWireTap Standard properties

These properties are used to configure cWireTap running in the Standard Job framework.

The Standard cWireTap component belongs to the Routing family.

Basic settings

URI	The endpoint URI to send the wire tapped message.
Populate new exchange	Select this check box to populate a new message to the wiretap URI.
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 from **None**, **Bean**, **Constant**, **CorrelationID**, **EL**, **Groovy**, **Header**, **JavaScript**, **JoSQL**, **JSonPath**, **JXPath**, **MVEL**, **OGNL**, **PHP**, **Property**, **Python**, **Ruby**, **Simple**, **SpEL**, **SQL**, **XPath**, and **XQuery**. **Expression TXT**: 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 create a copy of the original exchange, which will be the totally the same as the original one. If this check box is not selected, only a new exchange with the same endpoint name will be created. The message body and headers are null. The exchange pattern is **InOnly** for both conditions.

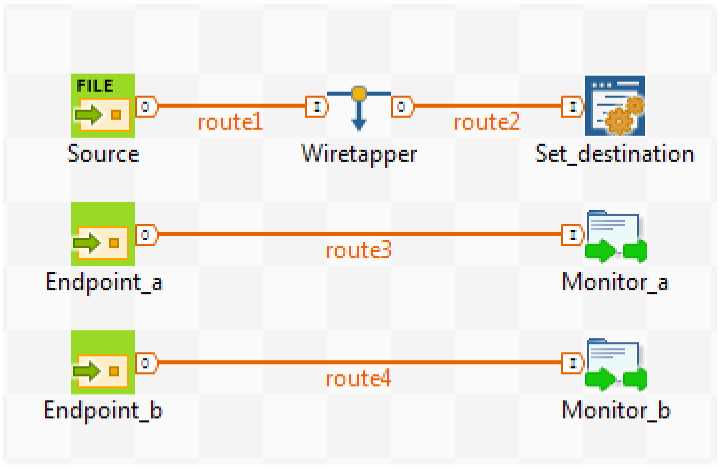
Usage

Usage rule	**cWireTap** can be a middle component in a Route.
Limitation	n/a

Scenario: Wiretapping a message in a Route

This scenario applies only to a Talend solution with ESB.

In this scenario, a **cWireTap** component is used to route a message to a separate endpoint while it is routed to the ultimate destination.



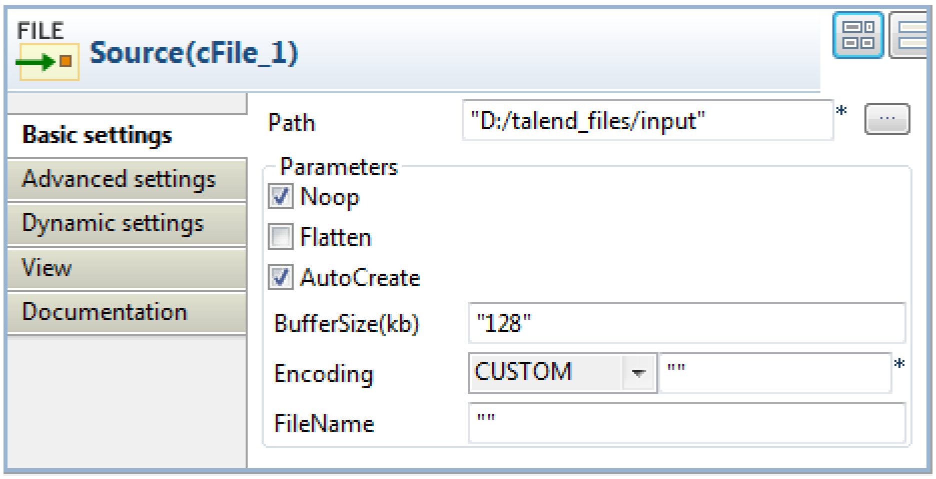
Dropping and linking the components

1. From the **Palette**, expand the **Connectivity** 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 **Core** folder, and drop a **cJavaDSLProcessor** components onto the design workspace.
4. Expand the **Custom** folder, and drop and two **cProcessor** components onto the design workspace.
5. Right-click the **cFile** component, select **Row** > **Route** from the contextual menu and click the **cWireTap** component.
6. Repeat this operation to connect the components as shown above.
7. 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.



FILE Source(cFile_1)

Basic settings | Advanced settings | Dynamic settings | View | Documentation

Path "D:/talend_files/input"

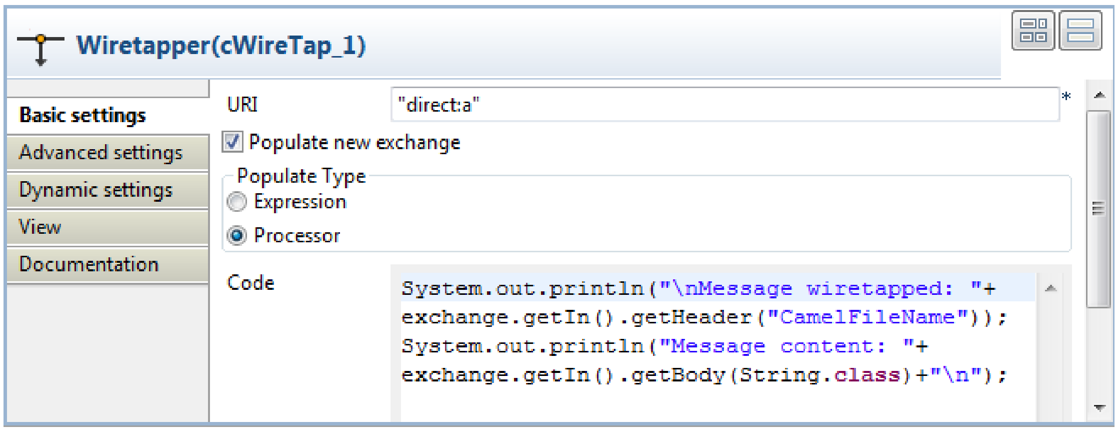
Parameters
- [✓] Noop
- [ ] Flatten
- [✓] AutoCreate

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.



Wiretapper(cWireTap_1)

Basic settings | Advanced settings | Dynamic settings | View | Documentation

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");
```

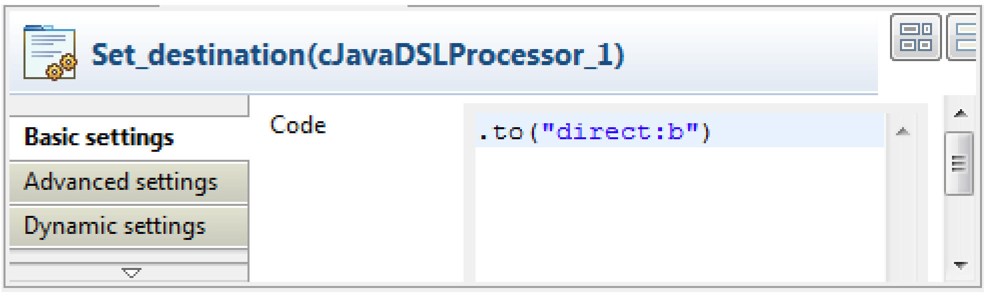
4. Enter `"direct:a"` in the **URI** field to route the wiretapped message to this endpoint.

   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");

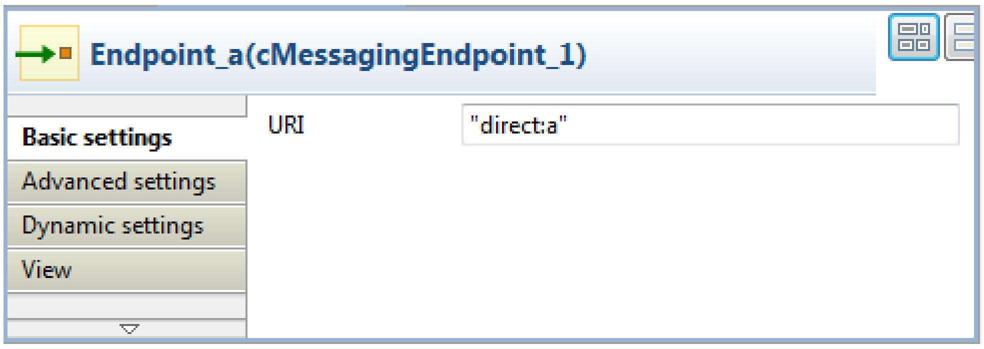
5. Double-click the **cJavaDSLProcessor** component to display its **Basic settings** view in the **Component** tab.



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.

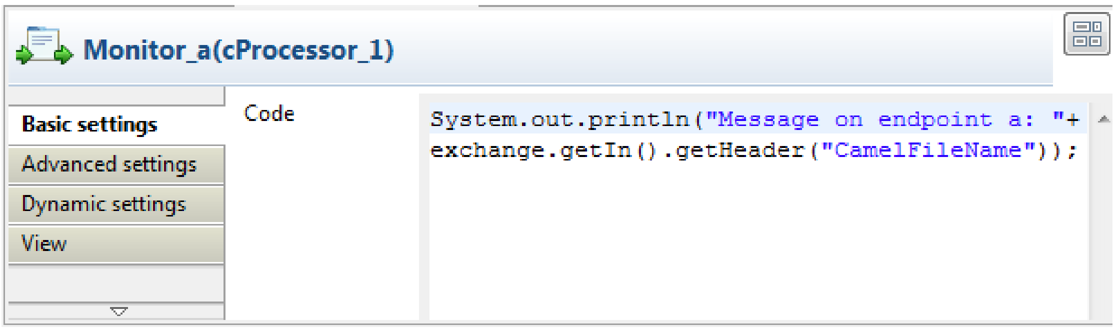


URI "direct:a"

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"));



Code

```
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

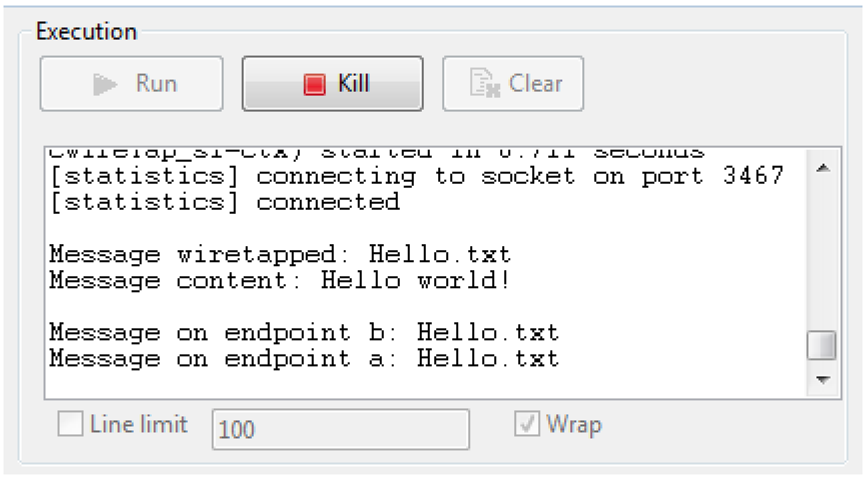
1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.

```
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*.



```
[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
```

cWMQ properties

cWMQ is used to exchange messages between a Route and a JMS provider using WMQ.

cWMQ sends messages to, or consumes messages from, a JMS Queue or Topic using the WebSphere broker.

cWMQ Standard properties

These properties are used to configure cWMQ running in the Standard Job framework.

The Standard cWMQ component belongs to the Connectivity/Messaging family.

Basic settings

URI/Type	Select the messaging type, either **queue** or **topic**.
URI/Destination	Type in a name for the message queue or topic in the message broker.
ConnectionFactory	Select a WMQ connection factory to be used for handling messages from the drop-down list.

Advanced settings

MQ Properties	Set the optional parameters in the corresponding table. Click **[+]** as many times as required to add parameters to the table. Then click the corresponding value field and enter a value. See the site http://camel.apache.org/jms.html for available options.

Usage

Usage rule	**cWMQ** can be a start, middle or end component in a Route. It has to be used with the **cMQConnectionFactory** component, which creates a connection to a MQ server. For more information about **cMQConnectionFactory**, see cMQConnectionFactory properties on page 146.
Limitation	Due to license incompatibility, one or more JARs required to use this component are not provided. You can install the missing JARs for

	this particular component by clicking the **Install** button on the **Component** tab view. You can also find out and add all missing JARs easily on the **Modules** tab in the **Integration** perspective of your studio. You can find more details about how to install external modules in Talend Help Center (https://help.talend.com). To run the Route using the **cWMQ** component in the studio, you need to download the *com.ibm.mq.jar*, *com.ibm.mq.commonservices.jar*, *com.ibm.mq.headers.jar*, *com.ibm.mq.jmqi.jar* and *connector.jar* from the IBM web site and add them to the **Dependencies** list of the **cMQConnectionFactory**. For more information about **cMQConnectionFactory**, see cMQConnectionFactory properties on page 146. To run the Route using the **cWMQ** component in *Talend Runtime* , before deploy the Route, you need to download *com.ibm.mq.osgi.java_7.5.0.5.jar* from the IBM web site and add it to the `<TalendRuntimePath>/container/deploy` folder. Alternatively, copy the *com.ibm.mq.jar*, *com.ibm.mq.commonservices.jar*, *com.ibm.mq.headers.jar*, *com.ibm.mq.jmqi.jar* and *connector.jar* to the `<TalendRuntimePath>/container/lib/ext` folder and change `<TalendRuntimePath>/container/etc/custom.properties`

	by adding the MQ packages to `org.osgi.framework.system.packages.extra`: org.osgi.framework.system.packages.extra = \ com.ibm.mq; \ com.ibm.mq.constants; \ com.ibm.mq.exits; \ com.ibm.mq.headers; \ com.ibm.mq.headers.pcf; \ com.ibm.mq.jmqi; \ com.ibm.mq.pcf; \ ...

Related scenario

For a related scenario, see Scenario 1: Sending and receiving a message from a JMS queue.

cMap properties

cMap executes transformations (called maps) between different sources and destinations by harnessing the capabilities of *Talend Data Mapper* , available in the **Mapping** perspective.

cMap transforms data from a wide range of sources to a wide range of destinations. If you want to use multiple inputs and/or outputs, you must use *Talend Data Mapper* I/O functions. For more information, see *Talend Data Mapper User Guide*.

cMap Standard properties

These properties are used to configure cMap running in the Standard Job framework.

The Standard cMap component belongs to the Transformation family.

The component in this framework is available when you have subscribed to one of the *Talend Platform* products or *Talend* Data Fabric.

Basic settings

Open Map Editor	Click the **[...]** button to open the map specified in the Map Path in the **Mapping** perspective.
Map Path	Specifies the map to be executed. Click the **[...]** button next to the **Map Path** field to open a dialog box in which you can select the map you want to use, then click the **[...]** button

	next to **Open Map Editor** to work with the map selected. Note that this map must have previously been created in the **Mapping** perspective.
Output Type	From the drop-down list, select how you want the output to be written. • **Default**: The default output is the same as the input, or Java if the map outputs Java. • **String**: Use this option if the data in the output column is to be a `String`. • **Byte Array**: Use this option if the data in the output column is to be a `Byte array`. • **InputStream**: Use this option if you are working with *Talend Data Mapper* metadata and the input is a stream.

Advanced settings

Log Level	From the drop-down list, select how often you want events to be logged. • **Infrequent**: Logs only events related to startup, shutdown and exceptions. • **Frequent** (default): Logs events related to startup, shutdown and exceptions, and once per map execution. • **Info**: Logs all events at an informational level or higher. • **All**: Logs all events. • **None**: Logs nothing.
Exception Threshold	*Talend Data Mapper* returns an execution status with an severity value which can be **OK**, **Info**, **Warning**, **Error** or **Fatal**. By setting the exception threshold, you can specify the severity level at which an exception is thrown, thus enabling downstream components to detect

	the error in cases other than the default value of **Fatal**. From the drop-down list, select the severity level at which an exception may be thrown during the execution of a map. • **Fatal** (default): An exception is thrown when a fatal error occurs. • **Error**: An exception is thrown when an error (or higher) occurs. • **Warning**: An exception is thrown when a warning (or higher) occurs.

Usage

Usage rule	**cMap** is used for Routes that require data mapping.
Limitation	Due to license incompatibility, one or more JARs required to use this component are not provided. You can install the missing JARs for this particular component by clicking the **Install** button on the **Component** tab view. You can also find out and add all missing JARs easily on the **Modules** tab in the **Integration** perspective of your studio. You can find more details about how to install external modules in Talend Help Center (https://help.talend.com). This limitation applies to maps that reference databases.

Note:

For further information about performing transformations using *Talend Data Mapper* , see *Talend Data Mapper User Guide*.