Contents

Copyleft ....................................................................................................................... 4
cAggregate .................................................................................................................. 6
cAMQP ...................................................................................................................... 16
cAWSConnection ...................................................................................................... 24
cAWSS3 .................................................................................................................... 25
cAWSSES ................................................................................................................. 34
cAWSSNS ................................................................................................................. 42
cAWSSQS ................................................................................................................. 47
cBean ......................................................................................................................... 58
cBeanRegister ............................................................................................................ 59
cConfig ...................................................................................................................... 60
cContentEnricher ....................................................................................................... 65
cConvertBodyTo ....................................................................................................... 73
cSOAP ....................................................................................................................... 79
cREST ....................................................................................................................... 98
cDataset ................................................................................................................... 110
cDelayer .................................................................................................................. 113
cDirect ..................................................................................................................... 117
cDirectVM ............................................................................................................... 117
cDynamicRouter ...................................................................................................... 122
cErrorHandler .......................................................................................................... 128
cExchangePattern .................................................................................................... 133
cFile ......................................................................................................................... 138
cFlatPack ................................................................................................................. 141
cFtp ........................................................................................................................ 150
cHttp ........................................................................................................................ 151
cIdempotentConsumer ............................................................................................. 162
cIntercept ................................................................................................................. 167
cJavaDSLProcessor ................................................................................................. 172
cJMS ........................................................................................................................ 172
cKafka ..................................................................................................................... 191
cLoadBalancer...................................................... 199
cLog................................................................. 205
cLoop............................................................... 206
cMail................................................................. 208
cMessageFilter................................................. 212
cMessageRouter................................................. 217
cMessagingEndpoint................................. 223
cMock.............................................................. 244
cMQConnectionFactory................................. 246
cMQTT.............................................................. 252
cMulticast....................................................... 260
cOnException.................................................. 266
cPipesAndFilters............................................. 272
cProcessor....................................................... 275
cRecipientList.................................................. 276
cRoutingSlip.................................................... 282
cSEDA.............................................................. 290
cSetBody.......................................................... 302
cSetHeader....................................................... 305
cSetProperty.................................................... 312
cSplitter.......................................................... 313
cStop............................................................... 321
cTalendJob....................................................... 321
cThrottler........................................................ 331
cTimer............................................................. 336
cTry................................................................. 337
cVM................................................................. 342
cWireTap........................................................ 344
cWMQ............................................................. 350
Adapted for 6.5.1. Supersedes previous releases.

Publication date: January 18, 2018

This documentation is provided under the terms of the Creative Commons Public License (CCPL).

For more information about what you can and cannot do with this documentation in accordance with the CCPL, please read: http://creativecommons.org/licenses/by-nc-sa/2.0/.

**Notices**

Talend and Talend ESB are trademarks of Talend, Inc.

Talend, Talend Integration Factory, Talend Service Factory, and Talend ESB are trademarks of Talend, Inc.


All brands, product names, company names, trademarks and service marks are the properties of their respective owners.

**License Agreement**

The software described in this documentation is licensed under the Apache License, Version 2.0 (the “License”); you may not use this software except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0.html. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

XmlSchema Core, Xmlsec - Apache Santuario, Zip4J, atinject, dropbox-sdk-java: Java library for the Dropbox Core API, google-guice. Licensed under their respective license.
cAggregate

Combines a number of messages together into a single message.

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

<table>
<thead>
<tr>
<th>Language</th>
<th>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, JXPath, 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 on page 79.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correlation expression/Expression</td>
<td>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 on page 79.</td>
</tr>
<tr>
<td>Correlation expression/Add Namespaces</td>
<td>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</td>
</tr>
</tbody>
</table>
namespaces as required to the table and define the prefix and URI in the corresponding columns.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Specify a Java bean to use as the aggregation strategy.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completion conditions/Number of messages</td>
<td>Select this check box to specify the number of messages to aggregate per batch before the aggregation is complete.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong></td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Completion conditions/Inactivity timeout (in milliseconds)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong></td>
</tr>
<tr>
<td></td>
<td>You can not use this option together with Scheduled interval. Only one of them can be used at a time.</td>
</tr>
<tr>
<td>Completion conditions/Scheduled interval (in milliseconds)</td>
<td>Select this check box to specify a repeating period (in milliseconds) by which the aggregator will complete all current aggregated exchanges.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong></td>
</tr>
<tr>
<td></td>
<td>You cannot use this option together with Inactivity timeout. Only one of them can be used at a time.</td>
</tr>
<tr>
<td>Completion conditions/Predicate matched</td>
<td>Select this check box to specify a predicate to indicate when an aggregated exchange is complete.</td>
</tr>
<tr>
<td>Completion conditions/Batch consumer</td>
<td>Select this check box to aggregate all files consumed from a file endpoint in a given poll.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Use persistence</th>
<th>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.</th>
</tr>
</thead>
</table>
| 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 Persistant 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. |
Scenario: Aggregating three messages into one

This scenario applies only to a Talend solution with ESB.

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

Creating a Java bean as the aggregation strategy

**About this task**

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

**Procedure**

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

2. The [New Bean] wizard opens. In the **Name** field, type in a name for the bean, for example, `AggregateBody`. Click **Finish** to close the wizard.
3. Type in the codes as shown in the figure below. In this use case, we just want to aggregate all messages into a single message.

```java
package beans;

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

public class AggregateBody implements AggregationStrategy{

    public Exchange aggregate(Exchange oldEx, Exchange newEx) {
        if(oldEx==null){
            return newEx;
        }
        String oldBody = oldEx.getIn().getBody(String.class);
        String newBody = newEx.getIn().getBody(String.class);
        newEx.getIn().setBody(oldBody+newBody);
        return newEx;
    }
}
```
4. Press Ctrl+S to save your bean.

Dropping and linking the components

About this task

![Diagram showing the components and connections]

Procedure

1. From the Palette, expand the Connectivity folder, and drop a cFile component onto the design workspace.
2. Expand the Routing folder, and drop a cAggregate component onto the design workspace.
3. Expand the Custom folder, and drop two cProcessor components onto the design workspace.
4. Right-click the cFile component, select Row > Route from the contextual menu and click the first cProcessor component.
5. Repeat this operation to connect the first cProcessor component to the cAggregate component.
6. Right-click the cAggregate component, select Row > Aggregate from the contextual menu and click the second cProcessor component.
7. Label all the components to better identify their functionality, as shown above.

Configuring the components

Procedure

1. Double-click the cFile component, which is labelled File_source, to display its Basic settings view in the Component tab.
2. In the Path field, browse to or enter the input file path, and leave the other parameters as they are. In this scenario, there are four text files in the specified directory: a.txt, b.txt, c.txt and d.txt, the contents of which are This is a!, This is b!, This is c!, and This is d! respectively.

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

![Aggregate Component](image)

4. In the Language field, select Constant or Simple as the expression language. In the Expression field, enter the expression "getBody(String.class) " to retrieve the body of the message. In the Strategy field, enter the name of the Java bean AggregateBody you just created. Select the Number of messages check box and type in 2 in the field.

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

![Monitor_before Component](image)

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

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

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

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

8. Press Ctrl+S to save your route.

Viewing code and executing the Route

Procedure

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

```java
public void initRoute() throws Exception {
    RouteBuilder = new org.apache.camel.builder.RouteBuilder() {
        public void configure() throws Exception {
            from(uriMap.get("File_source"))
                .routeId("File_source")
                .process(new org.apache.camel.Processor() {
                    public void process(
                        org.apache.camel.Exchange exchange)
                        throws Exception {
                            System.out
                                .println("Before aggregation: 
                                    + exchange
                                .getIn()
                                .getBody(
                                    String.class));
                    }
                })
                .aggregate(
                    simple("getBody(String.class)")
                .new beans.AggregateBody())
                .completionTimeout(1000)
                .completionFromBatchConsumer().id("aggregate_1")
                .process(
                    new org.apache.camel.Processor() {
                        public void process(
                            org.apache.camel.Exchange exchange)
                            throws Exception {
                                System.out
                                    .println("After aggregation: 
                                        + exchange
                                .getIn()
                                .getBody(
                                    String.class));
                            }
                        }
                    })
                );
}
```

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

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

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

Exchange messages between a Route and a JMS provider 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

<table>
<thead>
<tr>
<th>URI/Type</th>
<th>Select the messaging type, either queue or topic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI/Destination</td>
<td>Type in a name for the message queue or topic in the message broker.</td>
</tr>
<tr>
<td>ConnectionFactory</td>
<td>Click [...] and select an MQ connection factory to be used for handling messages.</td>
</tr>
</tbody>
</table>

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` on page 246.

**Limitation**

- n/a

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

This scenario applies only to a Talend solution with ESB.

This scenario will show you how to use the `cAMQP` component to send messages to and consume messages from a JMS Queue or Topic. To do this, two Routes are built, a message producer Route, and a consumer Route. Messages are sent to the AMQP 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 AMQP 1.0 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

About this task

Procedure

1. From the Palette, drag and drop a cMQConnectionFactory, a cTimer, a cSetBody, a cAMQP, 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

Procedure

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

2. From the MQ Server list, select AMQP 1.0 to handle messages. Keep the default settings of the other options.
3. Double-click the cTimer component to open its Basic settings view in the Component tab.
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.

6. Select **SIMPLE** from the **Language** drop-down list and type in "Hello world" in the **Expression** field as the message body.

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

8. From the **Type** list, select **queue** or **topic** to send the messages to a JMS queue or topic. In this use case, **queue** is used.
   
   In the **Destination** field, type in a name for the queue, for example "myqueue".
   
   In the **ConnectionFactory** field, click [...] and select the MQ connection factory that you have just configured to handle messages.

9. Keep the default settings of the **cLog** component to log the message exchanges.
10. Press Ctrl+S to save your Route.

**Viewing the code and executing the Route**

**Procedure**

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

   ```java
   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("cAMQP_1")).id("cAMQP_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 cAMQP_1 and cLog_1.

2. Press F6 to execute the Route. The logs of the message exchange are printed in the console.
3. In the ActiveMQ Web Console, you can see that the message queue `myqueue` has been created.

![ActiveMQ Console Screenshot]

### Building the consumer Route

**Adding components to arrange the flow**

**About this task**

![Diagram of consumer Route]

**Procedure**

1. From the Palette, drag and drop a `cMQConnectionFactory`, a `cAMQP`, 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

Procedure

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

   ![cMQConnectionFactory Component](image1.png)

   - MQ Server: AMQP 1.0
   - Host: "localhost"
   - Port: "5672"

2. From the **MQ Server** list, select **AMQP 1.0** to handle messages. Keep the default settings of the other options.

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

   ![Consumer(cAMQP_1) Component](image2.png)

   - URI Type: queue
   - Destination: "myqueue"
   - Connection Factory: cMQConnectionFactory_1

4. Specify the same URI **Type** and **Destination** in the consumer **cAMQP** component as in the producer.

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

5. Keep the default settings of the **cLog** component to log the message exchanges.

   ![logMessage(cLog_1) Component](image3.png)

   - Logging Category: "cAMQP_consumer.cLog_1"
   - Level: INFO
   - Options: NONE
6. Press **Ctrl+S** to save your Route.

**Executing the Route**

**Procedure**

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("cAMQP_1")).routeId("Consumer_cAMQP_1")
           .to(uriMap.get("cLog_1"))
           .id("cLog_1");
   }
   ``

   As shown above, the message flow is routed from `cAMQP_1` to `cLog_1`.

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

3. In the ActiveMQ Web Console, you can see that the messages in `myqueue` have been consumed.
cAWSConnection

Establishes a connection to Amazon Web Services for data storage and retrieval.

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Key</td>
<td>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).</td>
</tr>
<tr>
<td>Secret Key</td>
<td>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.</td>
</tr>
</tbody>
</table>
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 28
- Scenario: Sending Email using the cAWSSES component on page 36
- Scenario: Sending messages to Amazon's SNS topic on page 43
- Scenario: Sending messages to and receiving messages from Amazon's SQS queue on page 50

**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

<table>
<thead>
<tr>
<th>Connection</th>
<th>Select an AWSS3 connection component from the list to reuse the connection details you already defined.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bucket Name</td>
<td>Specify the name of the bucket, namely the top level folder, on Amazon's S3 service.</td>
</tr>
<tr>
<td></td>
<td>The following options are available only when the cAWSS3 is used as a Producer:</td>
</tr>
<tr>
<td>Storage Class</td>
<td>Select from <strong>Standard</strong>, <strong>Standard - Infrequent Access</strong> and <strong>Reduced Redundancy</strong>. For more information about storage classes, see the site <a href="http://docs.aws.amazon.com/AmazonS3/latest/dev/storage-class-intro.html">http://docs.aws.amazon.com/AmazonS3/latest/dev/storage-class-intro.html</a>.</td>
</tr>
<tr>
<td>Delete After Write</td>
<td>Select this check box to delete the local file object after it is uploaded to the S3 service.</td>
</tr>
<tr>
<td>Multi Part Upload</td>
<td>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.</td>
</tr>
<tr>
<td>Server Side Encryption</td>
<td>Select this option to encrypt the object on the server side using the AWS-managed keys.</td>
</tr>
<tr>
<td>Message Headers</td>
<td>Set the message headers in the corresponding table. Click <code>[+]</code> as many times as required to add message headers to the table. Then click the corresponding <strong>Value</strong> field and enter a value. See the site <a href="http://camel.apache.org/aws-s3.html">http://camel.apache.org/aws-s3.html</a> for available headers. Be sure to set the <code>CamelAwsS3Key</code> header which will be used as the name of the S3 file.</td>
</tr>
</tbody>
</table>
### 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:

<table>
<thead>
<tr>
<th><strong>File Name</strong></th>
<th>Specify the name of the file to be consumed.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Max Messages Per Poll</strong></td>
<td>The maximum number of objects that can be retrieved in one poll.</td>
</tr>
<tr>
<td><strong>Prefix</strong></td>
<td>Specify the prefix of files so that only files with that prefix will be consumed.</td>
</tr>
<tr>
<td><strong>Delete After Read</strong></td>
<td>Select this check box to delete the object from the S3 service after it has been retrieved.</td>
</tr>
<tr>
<td><strong>Include Body</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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](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 on page 24. |
| 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

About this task

Procedure

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

Procedure

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

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.

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.
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

Procedure

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

```java
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 \texttt{s3producer\_cFile\_1} to \texttt{s3producer\_cAWSS3\_1}.

2. Press \texttt{F6} to execute the Route. The logs of the message exchange are printed in the console.

3. In the Amazon's S3 Web Console, you can see that the object \texttt{talend.txt} has been created.
Building the consumer Route

Arranging the flow of the message

About this task

Procedure

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

Procedure

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.
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.

5. In the **Code** box, enter the following code to print the file name and its content in the execution console:

   ```java
   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**

**Procedure**

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

---

**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

<table>
<thead>
<tr>
<th>Connection</th>
<th>Select an AWS connection component from the list to reuse the connection details you already defined.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>Specify the email subject. It can be overridden by the CamelAwsSesSubject header.</td>
</tr>
<tr>
<td>From</td>
<td>Specify the sender's email address. It can be overridden by the CamelAwsSesFrom header.</td>
</tr>
<tr>
<td>To</td>
<td>Specify one or more destination email addresses. It can be overridden by the CamelAwsSesTo header.</td>
</tr>
<tr>
<td>Return Path</td>
<td>Specify the email address to which bounce notifications are to be forwarded. It can be overridden by the CamelAwsSesReturnPath header.</td>
</tr>
<tr>
<td>Reply to Addresses</td>
<td>Specify the reply-to email address(es) for the message. It can be overridden by the CamelAwsSesReplyToAddresses header.</td>
</tr>
<tr>
<td>Message Headers</td>
<td>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 <a href="http://camel.apache.org/aws-ses.html">http://camel.apache.org/aws-ses.html</a> for available headers.</td>
</tr>
</tbody>
</table>

### 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 on page 24. |
| Limitation | n/a |

Scenario: Sending Email using the cAWSSES component

This scenario applies only to a Talend solution with ESB.

This scenario will show you how to use the cAWSSES component to send emails with Amazon's SES service.

Dropping and linking the components

About this task

![Diagram of the components](image)

Procedure

1. From the Palette, drag and drop a cAWSConnection, a cTimer, a cSetBody, a cAWSSES, 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 the components

Procedure

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

2. In the Access Key and Secret Key fields, enter the authentication credentials.

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

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.

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 cAWSSES component to open its Basic settings view in the Component tab.
8. In the **Connection** list, select the **cAWSConnection** component that you have just configured to connect to Amazon SES service.
   
   In the **Subject** field, enter the email subject, for example, "Hello" in this use case.
   
   In the **From** and **To** fields, enter the sender's email address and the destination email address respectively.
   
   In the **Return Path** field, enter the email address to which bounce notifications are to be forwarded.
   
   In the **Reply To** field, enter the email address to receive replies.

9. Double-click the **cProcessor** component to display its **Basic settings** view in the **Component** tab.
10. In the Code box, enter the following code to print the CamelAwsSesMessageId in the execution console:

```java
System.out.println("CamelAwsSesMessageId: "+exchange.getIn().getHeader("CamelAwsSesMessageId"));
```

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

**Viewing the code and executing the Route**

**Procedure**

1. Click the Code tab at the bottom of the design workspace to check the generated code.
As shown above, the message flow from `sesproducer_cTimer_1` is given a payload by `sesproducer_cSetBody_1`. Then it is sent to `sesproducer_cAWSSES_1` and processed by `sesproducer_cProcessor_1`.

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

3. An email is sent to the destination address.
4. When replying the email, the destination is the email address specified in the **To** field of the cAWSSES component.
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

<table>
<thead>
<tr>
<th>Connection</th>
<th>Select an AWS connection component from the list to reuse the connection details you already defined.</th>
</tr>
</thead>
</table>

-----Original Message-----
From: noreply@talend.com [mailto:noreply@talend.com]
Sent: Monday, October 10, 2016 5:02 PM
To: Ning Wang <nwang@talend.com>
Subject: Hello

Hello World!
<table>
<thead>
<tr>
<th>Topic</th>
<th>Specify the name of the topic on Amazon's Simple Notification Service (SNS) to send message to.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>Specify a subject for the message. It can be overridden by the message header CamelAwsSnsSubject.</td>
</tr>
</tbody>
</table>

**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 on page 24.

**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**

**About this task**
Procedure

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

Procedure

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

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.

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.
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.

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

Procedure

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

```python
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"  
   + "?amazonSNSClientShared=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.
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.

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

<table>
<thead>
<tr>
<th>Connection</th>
<th>Select an AWS connection component from the list to reuse the connection details you already defined.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue Name</td>
<td>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</td>
</tr>
</tbody>
</table>
The following options are available only when the **cAWSSQS** is used as a Producer:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Delay (in seconds)</strong></td>
<td>Specify the amount of time to delay the first delivery of all messages added to this queue.</td>
</tr>
<tr>
<td><strong>Wait Time (0 to 20 seconds)</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

The following options are available only when the **cAWSSQS** is used as a Consumer:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Delete Message / After Read (processed by route)</strong></td>
<td>Select this check box to delete the message from the queue after it is read and processed by the Route.</td>
</tr>
<tr>
<td><strong>Delete Message / If Filtered (matched by filter)</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Allow Multiple Threads</strong></td>
<td>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 <strong>Concurrency Number</strong> and <strong>Max Messages Per Poll</strong> fields respectively.</td>
</tr>
<tr>
<td><strong>Extend Message Visibility</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
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](http://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/API_ChangeMessageVisibility.html).

<table>
<thead>
<tr>
<th>Request Attribute / All</th>
<th>Select this check box to retrieve the standard Amazon SQS attributes along with each message. For more information about the Amazon SQS attributes, see <strong>ReceiveMessage &gt; Request Parameters</strong>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Attribute / Approximate First Receive Timestamp</td>
<td>Select this check box to retrieve the <code>ApproximateFirstReceiveTimestamp</code> attribute only along with each message. For more information about the Amazon SQS attributes, see <strong>ReceiveMessage &gt; Request Parameters</strong>.</td>
</tr>
<tr>
<td>Request Attribute / Approximate Receive Count</td>
<td>Select this check box to retrieve the <code>ApproximateReceiveCount</code> attribute only along with each message. For more information about the Amazon SQS attributes, see <strong>ReceiveMessage &gt; Request Parameters</strong>.</td>
</tr>
<tr>
<td>Request Attribute / Sender ID</td>
<td>Select this check box to retrieve the <code>SenderId</code> attribute only along with each message. For more information about the Amazon SQS attributes, see <strong>ReceiveMessage &gt; Request Parameters</strong>.</td>
</tr>
<tr>
<td>Request Attribute / Sent Timestamp</td>
<td>Select this check box to retrieve the <code>SentTimestamp</code> attribute only along with each message. For more information about the Amazon SQS attributes, see <strong>ReceiveMessage &gt; Request Parameters</strong>.</td>
</tr>
</tbody>
</table>
Advanced settings

<table>
<thead>
<tr>
<th>AWS Account ID (Queue Owner)</th>
<th>Specify the queue owner’s AWS account ID when you need to connect the queue with different account owner.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue Configuration / Attributes</td>
<td>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 <a href="http://camel.apache.org/aws-sqs.html">http://camel.apache.org/aws-sqs.html</a> for available options.</td>
</tr>
<tr>
<td>Request Message Attribute</td>
<td>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.</td>
</tr>
</tbody>
</table>

Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cAWSSQS can be a start, middle or end component in a Route. It has to be used with the cAWSCConnection component, which creates a connection to Amazon SNS service. For more information about cAWSCConnection, see cAWSCConnection on page 24.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

About this task

![Diagram of producer route with components and connections]

Procedure

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

Procedure

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

   ![Basic settings view for cAWSConnection]

   - **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).

2. Double-click the **cTimer** labelled **repeat=2** to open its **Basic settings** view in the Component tab.
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.

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.

8. Configure the **cSetBody** labelled *body=world* in the same way to set the message body as *world*. 
9. Double-click the cAWSSQS labelled *Producer_1* to open its **Basic settings** view in the **Component** tab.

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

**Procedure**

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

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

Arranging the flow of the message

About this task

Procedure

1. From the Palette, drag and drop a **cAWSSConnection**, 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

Procedure

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.

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.

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

**Executing the Route**

**Procedure**

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

   ```
from "aws-sqs:talend-com-test-camel-sqs"
   + ?awsSqsClient=#AWSQS_AWSConnection_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.
3. In the SQS Web Console, you can see that there are now 3 three messages left in the queue `talend-com-testb-camel-sqs`. The two messages with the body `hello` have been consumed and deleted as configured in the `cAWSQS` component.

![SQS Web Console Image]

**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*.

<table>
<thead>
<tr>
<th>Specify the method</th>
<th>Select this check box to enter the name of a method to be included in the bean.</th>
</tr>
</thead>
</table>

**Usage**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th><strong>cBean</strong> can be a start, middle or end component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Related Scenario**

For a related scenario, see:

- **cConvertBodyTo**: Scenario: Converting the body of an XML file into an *org.w3c.dom.Document.class* on page 74.

**cBeanRegister**

Registers a Java bean in the registry to be used 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**

<table>
<thead>
<tr>
<th>Id</th>
<th>Enter any string which is used to look up the bean in the registry.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>Select this option to call a bean class that is stored in the <em>Code</em> node of the <em>Repository</em>.</td>
</tr>
<tr>
<td>Customized</td>
<td>Select this option to define the Java bean by entering the code in the <em>Code</em> box.</td>
</tr>
<tr>
<td>Class Name</td>
<td>This field appears when the <strong>Simple</strong> option is selected.</td>
</tr>
</tbody>
</table>
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*.

<table>
<thead>
<tr>
<th>Specify Arguments</th>
<th>This check box appears when the <strong>Simple</strong> option is selected. Select this check box to set the optional arguments in the corresponding table. Click <strong>[+]</strong> as many times as required to add arguments to the table.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Imports</th>
<th>This box appears when the <strong>Customized</strong> option is selected. Enter the Java code that helps to import, if necessary, external libraries used in the <strong>Code</strong> box.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>This box appears when the <strong>Customized</strong> option is selected. Enter the code of the bean in the box.</th>
</tr>
</thead>
</table>

**Usage**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th><strong>cBeanRegister</strong> cannot be added directly in a Route.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Limitation</th>
<th>n/a</th>
</tr>
</thead>
</table>

**Related Scenario**

For a related scenario, see:

- **cConvertBodyTo**: Scenario: Converting the body of an XML file into an `org.w3c.dom.Document.class` on page 74.

**cConfig**

Sets the CamelContext using Java code.

**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

<table>
<thead>
<tr>
<th><strong>Imports</strong></th>
<th>Enter the Java code that helps to import, if necessary, external libraries used in the <strong>Code</strong> box.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code</strong></td>
<td>Write a piece of code to manipulate the CamelContext.</td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>Click [+] to add the library or libraries that are required by the CamelContext or Typeconverter Registry to the Studio. Click [...] in the <strong>Lib Path</strong> field to show the <strong>Select Module</strong> dialog box. Select the inner module from the list or browse to the external module of your choice and click <strong>OK</strong> to close the dialog box.</td>
</tr>
</tbody>
</table>

### Advanced settings

<table>
<thead>
<tr>
<th><strong>Use MDC Logging</strong></th>
<th>Select this check box to enable the use of MDC logging.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>To be able to use this option, you need to open the <strong>Log4j</strong> view in the [Project Settings] dialog box by clicking <strong>File &gt; Edit Project Properties</strong> on the toolbar of the Studio main window.</td>
</tr>
<tr>
<td></td>
<td>Select the <strong>Activate log4j in components</strong> check box to activate the log4j feature. Then change the <strong>ConversionPattern</strong> in the log4j template to add your custom MDC property like %X{mdcPropertyName}.</td>
</tr>
<tr>
<td></td>
<td>To show the MDC information in the log, use the <strong>cLog</strong> component and a <strong>cProcessor</strong> before and after the <strong>cLog</strong>. In the <strong>cProcessor</strong> before the <strong>cLog</strong>, enter the code <code>org.apache.log</code></td>
</tr>
</tbody>
</table>
4j.MDC.put("Key", "Value"); to add
the custom property. In the cProcessor after
the cLog, enter the code org.apache.log
4j.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 on page 205.

Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cConfig cannot be added directly in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

Procedure

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

Procedure

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.

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

```java
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.
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.

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

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

7. Press Ctrl+S to save your route.

Viewing code and executing the Route

Procedure

1. Click the Code tab at the bottom of the design workspace to have a look at the generated code.
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
Message content: Message: <hello>world!</hello>
[- dataset:/foo] dataset:/foo
INFO Sent: 2 messages so far. Last group took: 15 milliseconds which is: 133.333 messages per second. average: 133.333
Message content: Message: <hello>world!</hello>
[- dataset:/foo] dataset:/foo
INFO Sent: 4 messages so far. Last group took: 0 milliseconds which is: 7 messages per second. average: 265.667
Message content: Message: <hello>world!</hello>
```

**cContentEnricher**

Uses 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource URI</td>
<td>This refers to the destination to which a message will be delivered if <strong>using a producer</strong> is selected; it refers to the source from which a message will be obtained if <strong>using a consumer</strong> is selected.</td>
</tr>
<tr>
<td>Using a producer</td>
<td>Select this check box to use a producer to provide additional data, that is to say sending a message to the defined URI.</td>
</tr>
<tr>
<td>Using a consumer</td>
<td>Select this check box to use a consumer to obtain additional data, that is to say requesting a message from the defined URI.</td>
</tr>
<tr>
<td>Use Aggregation Strategy</td>
<td>Select this check box to define the aggregation strategy for assembling the basic message and the additional data.</td>
</tr>
<tr>
<td>Specify timeout</td>
<td>This area appears when <strong>Using a consumer</strong> is selected. The timeout options are as follows:</td>
</tr>
<tr>
<td></td>
<td><strong>Wait until a message arrive</strong>: the component keeps waiting for a message.</td>
</tr>
<tr>
<td></td>
<td><strong>Immediately polls the message</strong>: the component immediately polls from the defined URI.</td>
</tr>
<tr>
<td></td>
<td><strong>Waiting at most until the timeout triggers</strong>: select this check box to type in a timeout value in Millis. The component waits for the message only within the defined time period.</td>
</tr>
</tbody>
</table>
Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cContentEnricher allows you to use a consumer or producer to obtain additional data, respectively intended for event message messaging and request/reply messaging.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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.

```html
docs/introduction.html
docs/setup.html
```

Dropping and linking the components

**Procedure**

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

Procedure

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

![URLList(cFile_1)](image)

   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.

![cSplitter_1](image)

   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.
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.

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.

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.

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

**Configuring the second sub-route**

**Procedure**

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

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.
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.

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.
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](http://camel.apache.org/http4.html).

Viewing code and executing the Route

**Procedure**

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

```java
public void configure() throws java.lang.Exception {
    from(uriMap.get("URLlist cFile_1")).routeId("URLlist cFile_1")
        .split(body(String.class).tokenize("\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.CamelHttpPost\}").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.
The messages from the list of URLs are saved in defined directory of the local file system.

**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

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cConvertBodyTo is used as a middle component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

This scenario applies only to a Talend solution with ESB.

In this scenario, a **cConvertBodyTo** component is used to convert the body of an XML file into an org.w3c.dom.Document class. Then a **cBean** component imports the org.w3c.dom.Document class, checks its content and prints out the root element name and the content of each `category` element.

The XML file is as follows:

```xml
<bookstore>
  <bookshelf>
    <category>Cooking</category>
    <quantity>100</quantity>
  </bookshelf>
  <bookshelf>
    <category>Languages</category>
    <quantity>200</quantity>
  </bookshelf>
  <bookshelf>
    <category>Arts</category>
    <quantity>300</quantity>
  </bookshelf>
  <bookshelf>
    <category>Science</category>
    <quantity>400</quantity>
  </bookshelf>
</bookstore>
```

Creating a Bean

Procedure

1. From the repository tree view, expand the Code node and right click the Beans node. In the contextual menu, select Create Bean.
2. The [New Bean] wizard opens. In the Name field, type in a name for the bean, for example, \textit{PrintConvertToBean}. Click Finish to close the wizard.

3. Enter the following code in the design workspace.

```java
package beans;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
public class PrintConvertToBean {

    /**
     * print input message
     */
```
```java
* @param message
*/
public static void helloExample(Document message) {
    if (message == null) {
        System.out.println("There's no message here!");
        return;
    }
    Element rootElement = message.getDocumentElement();
    if (rootElement == null) {
        System.out.println("There's no root element here!");
        return;
    }
    System.out.println("The root element name is:");
    + rootElement.getNodeName());
    System.out.println("The book categories are:");
    NodeList types = rootElement.getElementsByTagName("category");
    for(int i = 0;i<types.getLength();i++){
        Element child = (Element) types.item(i);
        System.out.println(child.getFirstChild().getNodeValue());
    }
}
```

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

**Results**

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

**Dropping and linking the components**

**Procedure**

1. Drag and drop a **cFile**, a **cConvertBodyTo**, a **cBean** and a **cBeanRegister** from the **Palette** onto the workspace.
2. Link the **cFile**, **cConvertBodyTo** and **cBean** using the **Row > Route** connection as shown above.

3. Label the components to better identify their functionality.

**Configuring the components**

**Procedure**

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

   ![Image of cFile component settings](image)

   **Basic settings**
   - **Path**: D:/talend_files/input/books
   - **Encoding**: Custom
   - **fileName**: books.xml

2. In the **Path** field, enter or browse to the path to the source XML file.
   
   If the source file folder contains more than one file, enter the name of the XML file of interest in the **FileName** field, and leave the other parameters as they are.

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

   ![Image of cConvertBodyTo component settings](image)

   **Basic settings**
   - **Target Class Name**: org.w3c.dom.Document.class
4. In the **Target Class Name** field, enter your target class name, `org.w3c.dom.Document.class` in this scenario.

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

6. In the **Id** field, enter "bean".
   
   Select the **Simple** option and in the **Class Name** field, enter the name of the bean to be invoked, `beans.PrintConvertToBean` in this scenario.

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

8. Select **Reference** and in the **Id** field, enter "bean" to call the bean that is registered by the **cBeanRegister**.

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

**Viewing code and executing the Route**

**Procedure**

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.
As shown above, the message from the endpoint `Read_message_cFile_1` has its body converted to `org.w3c.dom.Document.class` by `cConvertBodyTo_1`, and then processed by bean invoked by `cBean_1`.

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

RESULT: The root element name and the contents of the **category** elements are displayed.

```
[statistics] connecting to socket on port 3759
[statistics] connected
The root element is: bookstore
The book categories are:
  Cooking
  Languages
  Arts
  Science
```

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

<table>
<thead>
<tr>
<th>Address</th>
<th>The service endpoint URL where the Web service is provided.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>In case cSOAP is used to consume a Web service and the endpoint lookup shall use the Service</td>
</tr>
</tbody>
</table>
Locate (the **Use Service Locator** check box is selected), the URL needs to be "locator://anyAddress/".

| 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. |

| 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 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.

<table>
<thead>
<tr>
<th>Service Class</th>
<th>This field appears when the <strong>serviceClass</strong> service type is selected. Enter the name of the service class to be used to provide the Web service.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dataformat</strong></td>
<td>The exchange data style. <strong>POJO</strong>, <strong>PAYLOAD</strong>, <strong>RAW</strong>, or <strong>CXF_MESSAGE</strong>.</td>
</tr>
<tr>
<td>POJOs (Plain Old Java Objects) are the Java parameters to the method being invoked on the target server.</td>
<td></td>
</tr>
<tr>
<td><strong>PAYLOAD</strong> is the message payload, the contents of the <strong>soap:body</strong>.</td>
<td></td>
</tr>
<tr>
<td><strong>RAW</strong> is the raw message that is received from the transport layer without SAM (Service Activity Monitor) support.</td>
<td></td>
</tr>
<tr>
<td><strong>CXF_MESSAGE</strong> is the raw message that is received from the transport layer with SAM support.</td>
<td></td>
</tr>
<tr>
<td><strong>Use GZip Compress</strong></td>
<td>Select this check box to compress the incoming messages into GZip format before sending.</td>
</tr>
<tr>
<td><strong>Service Name</strong></td>
<td>The service name this service is implementing. It maps to the <strong>wsdl:service@name</strong> in the format of <strong>ns:SERVICE_NAME</strong> where <strong>ns</strong> is a namespace prefix valid at this scope. This field gets filled</td>
</tr>
<tr>
<td><strong>Port Name</strong></td>
<td>The endpoint name this service is implementing. It maps to the <code>wsdl:port@name</code>, in the format of <code>ns:PORT_NAME</code> where <code>ns</code> is a namespace prefix valid at this scope. This field gets filled in automatically upon completion of the Service configuration.</td>
</tr>
<tr>
<td><strong>Allow Streaming</strong></td>
<td>This option appears when <strong>PAYLOAD</strong> is selected in the <strong>Dataformat</strong> list. Select this check box to keep the payload as a <code>javax.xml.transform.Source</code> object that would allow streaming over buffering. If this option is disabled, the <strong>cSOAP</strong> component will parse the incoming messages into DOM elements.</td>
</tr>
<tr>
<td><strong>Operation Name</strong></td>
<td>The operation name this service is implementing. It maps to the <code>wsdl:operation@name</code>, in the format of <code>ns:OPERATION_NAME</code> where <code>ns</code> is a namespace prefix valid at this scope. This option appears when the <strong>cSOAP</strong> component is used to consume a Web service. This field gets filled in automatically upon completion of the Service configuration.</td>
</tr>
<tr>
<td><strong>Use Service Registry</strong></td>
<td>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 <strong>cSOAP</strong> component is used to provide a Web service, the service deployed in Runtime will work with the service registry.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Use Authorization</th>
</tr>
</thead>
</table>

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.
| **Use Business Correlation** | 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*.

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](http://camel.apache.org/cxf.html) for available URI options.

### Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cSOAP can be a start, middle or end component in a Route.</th>
</tr>
</thead>
</table>

| 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](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. |
Scenario 1: Providing a Web service using cSOAP from a WSDL file

This scenario applies only to a Talend solution with ESB.

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

Dropping and linking the components

About this task

This use case requires one cSOAP component and one cProcessor component.

Procedure

1. From the Palette, expand the Connectivity/Services folder, and drop a cSOAP component onto the design workspace.
2. Expand the Custom folder, and drop a cProcessor component onto the design workspace.
3. Right-click the cSOAP component, select Row > Route from the contextual menu and click the cProcessor component.
4. Label the cSOAP component for better identification of its functionality.

Configuring the components

About this task

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

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

![Image of Basic settings view]

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

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

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

5. Click [...] next to Service Configuration to open the service configuration wizard. The WSDL field has been filled in with the selected WSDL file. Click 🔄 to show the available port in the Port Name box. Select the airportSoap port and click Finish to close the wizard. The Service Name and Port Name fields in the Basic settings view are filled in automatically.

![Image of service configuration wizard]

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

7. Press Ctrl+S to save your route.
Viewing code and executing the Route

Procedure

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

   ```java
   org.apache.cxf.endpoint.Endpoint_cSOAP_1 = getEndpoint(
   "cxf://" + "http://192.168.0.212:8000/airport.service"
   + "?dataFormat=PAYLOAD" + "&allowStreaming=false"
   + "&wsdlURL="
   + "D:\talend_files/input/airport_soap_route.wsdl"
   + "&serviceName="
   + "http://airportsoap.sopera.de\)airport"
   + "&portName="
   + "http://airportsoap.sopera.de\)airportSoap", true,
   false, false, (String[]) null);
   ```

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

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

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

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

This scenario applies only to a Talend solution with ESB.

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

Procedure

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

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

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

```java
package beans;

public interface CXFdemobean {
    public String helloExample(String message) ;
}
4. Press Ctrl+S to save your bean.

Dropping and linking the components

About this task

This use case requires one cSOAP component and one cProcessor component.

Procedure

1. From the Palette, expand the Connectivity/Services folder, select the cSOAP component and drop it onto the design workspace.
2. Expand the Custom folder, select the cProcessor component and drop it onto the design workspace.
3. Right-click the cSOAP component, select Row > Route in the contextual menu and click the cProcessor component.
4. Label the components for better identification of their functionality.

Configuring the components

About this task

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

Procedure

1. Double-click the cSOAP component to display its Basic settings view in the Component tab.
2. In the **Address** field, type in the service endpoint URL for the Web service to be provided, `http://192.168.0.212:8001/service.endpoint` in this example.

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

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

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

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

### Viewing code and executing the Route

**Procedure**

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

   ```java
   org.apache.camel.component.cxf.CxfEndpoint endpoint_cSOAP_1 = getCxfEndpoint(
       "cxf://" + "http://192.168.0.212:8001/service.endpoint",
       "?dataFormat=POJO" + "&loggingFeatureEnabled=true"
   + "&serviceClass=" + "beans.CXFdemobean", true, false,
   false, (String[]) null);
   
   As shown in the code, the **cSOAP** component produces the Web service from an predefined bean `beans.CXFdemobean` using the endpoint URL `http://192.168.0.212:8001/service.endpoint`.
   
2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route. You can also press **F6** to execute it.

   RESULT: The service is successfully started. You can access it from a Web browser using the service endpoint URL followed by `?wsdl`.
Scenario 3: Providing a Web service from a Route Resource

This scenario applies only to a Talend solution with ESB.

In this scenario, a Web service is provided from a Route Resource using a cSOAP component.

Creating a Route Resource

Procedure

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, DemoServiceWsdl. Click Finish to close the wizard.
3. Browse to an existing WSDL file from the local file system and click **Finish**.
4. Press **Ctrl+S** to save your Route Resource.

**Dropping and linking the components**

**About this task**

![Diagram showing cSOAP and cProcessor components](image)

This use case requires a **cSOAP** and a **cProcessor** component.

**Procedure**

1. From the **Palette**, expand the **Connectivity/Services** folder, and drop a **cSOAP** component onto the design workspace.
2. Expand the Custom folder, and drop a cProcessor component onto the design workspace.

3. Right-click the cSOAP component, select Row > Route from the contextual menu and click the cProcessor component.

4. Label the cSOAP component for better identification of its functionality.

Configuring the components

About this task

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

Procedure

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

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

3. From the Type list, select wsdlURI and use the Repository property type to start the Web service from the Route Resource that we have created.

4. In the WSDL File field, click the [...] button and select DemoServiceWsdl from the Resources tree view. Click OK to close the wizard.

5. Click [...] next to Service Configuration to open the service configuration wizard. The WSDL field has been filled in with the selected WSDL file. Click to show the available port in the Port Name box. Select the DemoServicePort port and click Finish to close the wizard. The Service Name and Port Name fields in the Basic settings view are filled in automatically.
6. Press **Ctrl+S** to save your Route.

**Viewing code and executing the Route**

**Procedure**

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

   ```java
   org.apache.camel.component.cxf.CxfEndpoint endpoint_cSOAP_1 = getEndpoint("cxf://" + "http://localhost:8000/service.endpoint" + "?dataFormat=PAYLOAD" + "&allowStreaming=false" + "&wsdlURL=" + getClass().getResource("/DemoServiceWsd1_0.1") + "\ntoString() + "&serviceName=" + "(http://www.talend.org/service/)DemoService" + "\nendpointName=" + "[http://www.talend.org/service/]DemoServicePort", true, false, false, (String[]) null);
   
   As shown in the code, the **cSOAP** component produces the Web service from an predefined Route Resource **DemoServiceWsd1_0.1** using the **endpoint URL** `http://localhost:8000/service.endpoint`.

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

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

Provides integration with Apache CXF for connecting to JAX-RS services.

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

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>The service endpoint URL where the REST service is provided.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Select which type you want to use to provide the REST service. Either <strong>Manual</strong> or <strong>resourceClass</strong>.</td>
</tr>
</tbody>
</table>

**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.

<p>| <strong>Operation</strong> | This field appears when the resourceClass service type is selected and cREST is used as the service consumer. Enter the name of the operation. |</p>
<table>
<thead>
<tr>
<th>Relative Path</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td>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. <strong>Warning:</strong> 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.</td>
</tr>
<tr>
<td>Content Type</td>
<td>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.</td>
</tr>
<tr>
<td>Accept Type</td>
<td>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</td>
</tr>
<tr>
<td><strong>Response Class</strong></td>
<td>This field appears when the <strong>Manual</strong> service type is selected and the <strong>cREST</strong> is used as the service consumer. Enter the name of the response class.</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Use Service Locator</strong></td>
<td>Select this check box to enable the Service Locator. Specify the service namespace and the service name in the corresponding fields.</td>
</tr>
<tr>
<td><strong>Enable the Service Activity Monitoring</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
| **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.  
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. |
| **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](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](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**

**About this task**

![Diagram of service provider Route]

**Procedure**

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

Procedure

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

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:

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

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.

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

Viewing code and executing the Route

Procedure

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

```java
from("http://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.rs.utils.CxfRsUtils.getStructuredParams((String)inMessage
            .getHeader(org.apache.camel.Exchange.HTTP_QUERY),
            "&", false, false));
    }
}).route("provider_cREST_1").setBody()
    .simple("ref:customers").id("provider_cSetBody_1")
    .to("logicRest.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.
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

About this task

Procedure

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

Procedure

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

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.

6. Press **Ctrl+S** to save your route.
Executing the Route

Procedure

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

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.
cDataset

Creates 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**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Id</strong></td>
<td>The ID of the Dataset bean.</td>
</tr>
<tr>
<td><strong>Produce Delay</strong></td>
<td>Specify a delay in milliseconds to cause producers to pause.</td>
</tr>
<tr>
<td><strong>Consume Delay</strong></td>
<td>Specify a delay in milliseconds to cause consumers to pause.</td>
</tr>
<tr>
<td><strong>Preload Size</strong></td>
<td>Specify how many messages should be sent before the Route completes its initialization.</td>
</tr>
<tr>
<td><strong>Initial Delay</strong></td>
<td>Specify the time in milliseconds to wait before starting sending messages.</td>
</tr>
<tr>
<td><strong>Minimum Rate</strong></td>
<td>Specify the least number of messages that the dataset should contain before starting sending messages.</td>
</tr>
<tr>
<td><strong>Register new Bean</strong></td>
<td>Select this check box to register a new bean.</td>
</tr>
<tr>
<td><strong>Bean Class</strong></td>
<td>Enter the class of the bean. This field appears when the Register new Bean check box is selected.</td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

**Usage**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th><strong>cDataset</strong> can be a start, middle, or end component of a Route.</th>
</tr>
</thead>
</table>
Scenario: Using cDataset to create messages

This scenario applies only to a Talend solution with ESB.

In this scenario, a `cDataset` component is used to create a DataSet instance as a source of messages.

Dropping and linking the components

Procedure

1. From the Testing folder, drag and drop a `cDataset` component onto the design workspace.
2. From the Core folder, drag and drop a `cSetBody` component onto the design workspace.
3. From the Miscellaneous folder, drag and drop two `cLog` components onto the design workspace.
4. Connect the components using Row > Route connections.
5. Label the components to better identify their roles in the Route, as shown above.

Configuring the components

Procedure

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

   ![Diagram of components](image)

2. In the Id field, enter "myDataset" as the dataset bean ID.
3. In the Arguments table, click [+] to add a row and enter 1000 in the Value field to create 1000 messages. Keep the default settings of the other options.
In the **Bean Class** field, the default implementation that is shipped by Camel `org.apache.camel.component.dataset.SimpleDataSet` is set. In this use case, a new dataset bean is registered.

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

![Set_message_body(cSetBody_1)](image)

5. Select **Constant** in the **Language** list and enter "Hello!" in the **Expression** field as the message body.

6. Keep the default settings of the **cLog** components to monitor the message exchanges.

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

**Viewing the code and executing the Route**

**Procedure**

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

```java
from("dataset:myDataset").routeId("cDataset_cDataset_1")
  .to("log:test.Log_1 + "?level=WARN")
  .id("cDataset_cLog_1").setBody().constant("Hello!")
  .id("cDataset_cSetBody_1")
  .to("log:test.Log_2 + "?level=WARN")
  .id("cDataset_cLog_2");
```

As shown in the code, the Route is built from the **cDataset_cDataset_1** endpoint, routed to the **cDataset_cLog_1**, set the message body as "Hello!" by the **cDataset_cSetBody_1**, and then routed to the **cDataset_cLog_2**.

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

A DataSet of 1000 messages is created and logged in the execution console. The default message body is `<hello>world!</hello>`. It is changed to **Hello!**.
**cDelayer**

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 |

**Scenario: Using cDelayer to delay message routing**

This scenario applies only to a Talend solution with ESB.

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

About this task

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

Procedure

1. From the Connectivity folder of the Palette, drop two cFile components onto the design workspace, one to read files from a local folder and the other to write the files to another local folder.
2. From the Custom folder of the Palette, drop two cProcessor components onto the design workspace, one next to the reading component to monitor messages read from the source file folder, and the other next to the writing component to monitor messages written to the target file folder.
3. From the Orchestration folder of the Palette, drop one cDelayer component onto the design workspace, between the message reading monitor component and the message writing component.
4. Connect the components using Row > Route connections.
5. Label the components to better identify their roles in the Route, as shown above.

Configuring the components

Procedure

1. Double-click the first cFile component, which is labelled Read, to open its Basic settings view in the Component tab.
2. In the **Path** field, enter or browse to the path to the source files, and leave the other parameters as they are.

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

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

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

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

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

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

7. Double-click the **cDelayer** component, which is labelled **Delay_timer**, to open its **Basic settings** view in the **Component** tab.
8. In the **Time to wait (in ms)** field, enter the number of milliseconds by which you want to delay message delivery. Note that the value must be a positive integer.
   In this use case, we want each message to be delivered after a 20-second delay.

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

**Viewing the code and executing the Route**

**Procedure**

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

   ```java
   public void initRoute() throws Exception {
       routeBuilder = new org.apache.camel.builder.RouteBuilder {
           public void configure() throws Exception {
               from(uriMap.get("Read")).routeId("Read").process(
                   new org.apache.camel.Processor () {
                       public void process(
                           org.apache.camel.Exchange exchange)
                           throws Exception {
                           Date date = new Date();
                           SimpleDateFormat formatter = new SimpleDateFormat("HH:mm:ss");
                           String s = formatter.format(date);
                           System.out.println("\nMessage \n" + exchange.getIn().getHeader("CamelFileName") + " read at " + s);
                       }
                   }).id("cProcessor_1").delay(20000).id("cDelay_1").to(uriMap.get("Write")).id("cFile_2") .process(
   ```

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

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

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

Produces and consumes messages synchronously in different threads within a single CamelContext.

**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**

<table>
<thead>
<tr>
<th>Name</th>
<th>This option appears when cDirect is used as a start component in a Route. Type in any string that uniquely identifies the endpoint.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Exist cDirect</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Usage**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cDirect is used as a start, middle, or end component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Related scenario:

For a related scenario, see Scenario: Using cSEDA, cVM and cDirect to produce and consume messages separately on page 293.

**cDirectVM**

Produces and consumes 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 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

<table>
<thead>
<tr>
<th>When using as a start component in a Route:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Endpoint Name</strong></td>
<td>Type in any string that uniquely identifies the endpoint.</td>
</tr>
<tr>
<td>When using as a middle or end component in a Route:</td>
<td></td>
</tr>
<tr>
<td><strong>Select From Existing(s)</strong></td>
<td>Click this radio button to select an existing consumer. Click [...] beside the <strong>Consumer</strong> field to show the existing consumer(s) and select the one to consume the message.</td>
</tr>
<tr>
<td><strong>Input Endpoint Name</strong></td>
<td>Click this radio button to enter the name of the consumer in the <strong>Endpoint Name</strong> field.</td>
</tr>
<tr>
<td><strong>Block if Consumer is not active</strong></td>
<td>Select this check box to let the producer block if the consumer is not active in the <strong>Timeout</strong> period.</td>
</tr>
<tr>
<td><strong>Timeout</strong></td>
<td>This option appears when the block is enabled. Specify the time in milliseconds before the producer stops waiting for the consumer to become active.</td>
</tr>
</tbody>
</table>

### Usage

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

### Scenario: Direct and synchronous messaging within the same JVM

This scenario applies only to a Talend solution with ESB.

In this scenario, a message exchange is triggered in one sub-route and synchronously consumed in another sub-Route using the Direct-VM mechanism. At the consumer end, the message content is processed and displayed on the console.
Dropping and linking the components

Procedure

1. From the Palette, drag and drop the following components onto the design workspace: a cTimer, a cSetHeader, a cSetBody, two cDirectVM, and a cProcessor.
2. Link the cTimer, the cSetHeader, the cSetBody and the first cDirectVM components using Row > Route connections to form a sub-route.
3. Link the other cDirectVM component to the cProcessor component using a Row > Route connection to form another sub-route.
4. Label the components to better identify their roles in the Route.

Configuring the components

Configuring the first sub-route

Procedure

1. Double-click the cTimer component (labelled Starter in this example) to display its Basic settings view on the Component tab.

2. In the Repeat field, enter 1 so that the message exchange is triggered only once. Leave the other settings as they are.
3. Double-click the `cSetHeader` component (labelled `Set_message_headers` in this example) to display its Basic settings view on the Component tab.

4. Click the [+] button to add two headers in the Headers table, and then give each header a name and a value.
   In this example, we name the headers Name and Company respectively, and enter Bill and Talend as their values.

5. Double-click the `cSetBody` component (labelled `Set_message_body` in this example) to display its Basic settings view on the Component tab.

6. Select SIMPLE from the Language list as we will trigger a simple text message exchange in this example.

7. In the Expression field, enter Hi there in double quotation marks as the content of the message body.

8. Double-click the first `cDirectVM` component (labelled `D-VM_producer` in this example) to display its Basic settings view on the Component tab.
Select an existing consumer endpoint, or specify the name of the consumer endpoint. In this example, we will name the consumer endpoint \textit{dvm}, so select the Input Endpoint Name option, and enter the name of the consumer endpoint \textit{dvm} in the Endpoint Name field.

Configuring the second sub-route

Procedure

1. Double-click the second \texttt{cDirectVM} component (labelled \textit{D-VM\_consumer} in this example) to display its Basic settings view on the Component tab.

2. In the Endpoint Name field, enter \textit{dvm} in double quotation marks to name the endpoint.

3. Double-click the \texttt{cProcessor} (labelled \textit{Show\_message} in this example) component to display its Basic settings view.

4. In the Code area, enter the following code to display the content of the message headers and the message body:

```java
exchange.getOut().setHeader("header1",exchange.getIn().getHeader("Name"));
exchange.getOut().setHeader("header2",exchange.getIn().getHeader("Company"));
exchange.getOut().setBody(exchange.getIn().getBody(String.class));
System.out.println();
System.out.println("Message for "+exchange.getOut().getHeader("header1")
+" of "+exchange.getOut().getHeader("header2")+":");
System.out.println(exchange.getOut().getBody(String.class));
```
Executing the Route

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

RESULT: The message generated in the first sub-route is consumed synchronously in the consumer endpoint and displayed on the console after processing. The message exchange is triggered once as configured.

cDynamicRouter

Routes a message or messages to different endpoints on specified conditions.

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

<table>
<thead>
<tr>
<th>Bean class</th>
<th>Enter the name of the bean class to be used for the dynamic router.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify the method</td>
<td>Select this check box to specify the method to be used which is defined in the bean class.</td>
</tr>
<tr>
<td>Ignore Invalid Endpoints</td>
<td>Select this check box to ignore unresolved endpoint URIs. Clear the check box to throw an exception when endpoint URIs are not valid.</td>
</tr>
</tbody>
</table>
Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>eDynamicRouter is used as a middle or end component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Scenario: Routing files conditionally to different file paths

This scenario applies only to a Talend solution with ESB.

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

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

**Message_1.xml**:

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

**Message_2.xml**:

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

**Message_3.xml**:

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

A predefined Java bean, `setDynaURI`, is called in this use case to return endpoint URIs according to the city name contained in each message, so that the message containing the city name Washington will be routed to endpoint Washington and so forth.
For more information about creating and using Java Beans, see *Talend Studio User Guide*.

```java
package beans;

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

public class setDynaURI {

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

About this task

![Diagram of dropping and linking components](image)

**Procedure**

1. From the **Palette**, expand the **Connectivity** folder, and drop one **cFile** and three **cMessagingEndpoint** components onto the design workspace.
2. Expand the **Routing** folder, and drop a **cDynamicRouter** component onto the design workspace.
3. Expand the **Custom** folder, and drop three **cProcessor** components onto the design workspace.
4. Label the components for better identification of their respective functionality.
5. Right-click the **cFile** component, select **Row > Route** from the contextual menu and click the **cDynamicRouter** component.
6. Repeat this operation to connect the **cMessagingEndpoint** components to the **cProcessor** components.

Configuring the components and connections

**Procedure**

1. Double-click the input **cFile** component to display its **Basic settings** view in the **Component** tab and set its properties.
   In this use case, simply specify the input file path and leave the other parameters as they are.
2. Double-click the `cDynamicRouter` component to display its Basic settings view in the Component tab.

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

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

   Here, we want to use this component to retrieve the message routed to the URI `direct:Washington`, as shown below.
5. Repeat this step to set the endpoint URIs for the other two `cMessagingEndpoint` components: `direct:London` and `direct:Beijing` respectively.

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

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

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

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

9. Press Ctrl+S to save your Route.

**Viewing code and executing the Route**

**Procedure**

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

```
public void initRoute() throws Exception {
    routeBuilder = new org.apache.camel.builder.RouteBuilder()
    public void configure() throws Exception {
        from(uriMap.get("Message_source")).
            dynamicRouter()
                bean(beans.setDynaURI.class)
                    id("cDynamicRouter_1");
        from(uriMap.get("Washington")).
            id("cInbound_1");
    } });
```

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

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

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

Processes errors in the message routing.

**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**

<table>
<thead>
<tr>
<th>Default Handler</th>
<th>This error handler does not support a dead letter queue and will return exceptions back to the caller.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Maximum Redeliveries: select this check box to set the number of redeliveries in the Maximum Redeliveries (int) field.</td>
<td></td>
</tr>
<tr>
<td>Set Redelivery Delay: select this check box to set the initial redelivery delay (in milliseconds) in the Redelivery Delay (long) field.</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td><strong>Asynchronous Delayed Redelivery</strong>: select this check box to allow asynchronous delayed redelivery.</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td></td>
</tr>
<tr>
<td><strong>More Configurations by Code</strong>: select this check box to enter codes in the Code box for further configuration.</td>
<td></td>
</tr>
<tr>
<td><strong>Dead Letter</strong></td>
<td>This handler supports attempting to redeliver the message exchange a number of times before sending it to a dead letter endpoint.</td>
</tr>
<tr>
<td><strong>Dead Letter Uri</strong>: 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.</td>
<td></td>
</tr>
<tr>
<td><strong>Logging Handler</strong></td>
<td>This handler logs the exceptions.</td>
</tr>
<tr>
<td><strong>Set Logger Name</strong>: select this check box to give a name to the logger in the Name field.</td>
<td></td>
</tr>
<tr>
<td><strong>Set Log Level</strong>: select this check box to decide the log level from the Level list.</td>
<td></td>
</tr>
</tbody>
</table>

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

This scenario applies only to a Talend solution with ESB.

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

Dropping and linking the components

\textbf{Procedure}

1. Drop the following components from the \texttt{Palette} onto the workspace: \texttt{cMessagingEndpoint}, \texttt{cErrorHandler} and \texttt{cProcessor}, labelled as \texttt{Jetty\_Server}, \texttt{Error\_Handler} and \texttt{Throw\_Exception} respectively.

2. Link \texttt{cMessagingEndpoint} and \texttt{cProcessor} using a \texttt{Row > Route} connection.

Configuring the components

\textbf{Procedure}

1. Double-click \texttt{cErrorHandler} to open its \texttt{Basic settings} view in the \texttt{Component} tab.

2. Select \texttt{Logging Handler} to log the exceptions that are thrown.

3. Double-click \texttt{cMessagingEndpoint} to open its \texttt{Basic settings} view in the \texttt{Component} tab.
4. In the **Uri** field, enter `jetty:http://localhost:8889/service` to specify the Jetty server.

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

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

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

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

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

**Viewing code and executing the Route**

**Procedure**

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.
As shown above, the route starts from the endpoint Jetty_Server and throws the exception of server side error via cProcessor_1.

2. Press F6 to execute the Route.

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

The Jetty server has started.

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

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

4. Go to the Studio and check the results in the Run tab.
As shown above, `cErrorHandler` has logged the exception at the level of `ERROR`.

### cExchangePattern

Sets the message exchange mode in a Route.

#### 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

Scenario: Enabling the InOut exchange pattern to get replies

This scenario applies only to a Talend solution with ESB.

In this scenario, a `cExchangePattern` component is used to enable the request/reply exchange pattern in the Route, so that the client can get a reply from the server.
To send requests to the server side, a soapUI is needed and its configuration will be briefed in the following contents.

To build the Route, do the following.

**Dropping and linking the components**

**Procedure**

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

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

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

**Configuring the components**

**Procedure**

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

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

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

4. In the **WSDL File** field, enter the URL of the wsdl file. You can also click the three-dot button to browse for it.
5. Click [...] next to Service Configuration to open the service configuration wizard. The WSDL field has been filled in with the selected WSDL file. Click ![checkmark] to show the available port in the Port Name box. Select the airportSoap port and click Finish to close the wizard. The Service Name and Port Name fields in the Basic settings view are filled in automatically.

6. In the Dataformat list, select PAYLOAD.

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

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

9. Double-click cProcessor to open its Basic settings view in the Component tab.
10. In the **Code** box, enter the code below.

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

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

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

**Viewing code and executing the Route**

**Procedure**

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.
As shown above, the route has its message exchange pattern set as InOut using the method .setExchangePattern(org.apache.camel.ExchangePattern.InOut). In the meantime, a string is created using StringBuilder sb = new StringBuilder() at cProcessor_1 and is used as the reply message via the method exchange.getOut().setBody(sb.toString()).

2. Press F6 to execute the Route.

The server Route gets started.

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

Procedure

1. In the soapUI, create a Test project and edit a request, as illustrated below:
Note that the wsdl file must be same as that configured for cSOAP, so that the request can be in line with the definition of the web service.

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

**cFile**

Provides access to file systems, allowing files to be processed by any other components or messages from other components to be saved to the disk.

**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

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Path</strong></td>
<td>Path to the file or files to be accessed or saved.</td>
</tr>
<tr>
<td><strong>Parameters/Noop</strong></td>
<td>Select this check box to keep the file or files in the original folder after being read.</td>
</tr>
<tr>
<td><strong>Parameters/Flatten</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Parameters/AutoCreate</strong></td>
<td>Select this check box to create the directory specified in the Path field automatically if it does not exist.</td>
</tr>
<tr>
<td><strong>Parameters/BufferSize(kb)</strong></td>
<td>Write buffer sized in bytes.</td>
</tr>
<tr>
<td><strong>Encoding</strong></td>
<td>Specify the encoding of the file, <strong>ISO-8859-15</strong>, <strong>UTF-8</strong>, or <strong>CUSTOM</strong>.</td>
</tr>
<tr>
<td><strong>FileName</strong></td>
<td>The name of the file to be processed. Use this option if you want to consume only a single file in the specified directory.</td>
</tr>
</tbody>
</table>

### Advanced settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advanced</strong></td>
<td>Set the optional arguments in the corresponding table. Click [+] as many times as required to add arguments to the table. Then click the corresponding <strong>Value</strong> field and enter a value. See the site <a href="http://camel.apache.org/file2.html">http://camel.apache.org/file2.html</a> for available URI options.</td>
</tr>
</tbody>
</table>
Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cFile can be a start, middle or end component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

This scenario applies only to a Talend solution with ESB.

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

Dropping and linking the components

Procedure

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

Configuring the components

Procedure

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

![Message_destination(cFile_2)](image)

4. In the **Path** field, browse to or enter the output file path, as shown above. Leave the other parameters as they are.
5. Press **Ctrl+S** to save your route.

**Viewing code and executing the Route**

**Procedure**

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

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

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

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

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

**cFlatPack**

Processes fixed width or delimited files or messages using 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](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 ",".

<p>| <strong>Split Rows</strong> | Select this check box to process each row one by one. |</p>
<table>
<thead>
<tr>
<th>Ignore First Record</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow Short Lines</td>
<td>Select this check box to allow lines shorter than expected by the PZMAP file.</td>
</tr>
<tr>
<td>Ignore Extra Columns</td>
<td>Select this check box to allow lines longer than expected by the PZMAP file and ignore the extra characters.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cFlatPack can be a start, middle, or end component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Scenario: Parsing delimited file using the cFlatPack component**

This scenario applies only to a Talend solution with ESB.

In this scenario, a cFile component reads a delimited file from the local file system, which contains the customers information including id, first name, surname, and order id, as shown below:

1,"Harry",Carter,21  
2,Padre,Boulevard,22  
3,Andrew,Polk,23  
4,Herbert,Reagan,"24"  
5,Chester,Eisenhower,25

The file is parsed by the cFlatPack component, using a predefined PZMAP XML file:

```xml
<?xml version="1.0"?>
```
The customer information is then printed in the execution console by a **cBean** component.

For more information about the PZMAP file configuration, see the website [http://flatpack.sourceforge.net/documentation/index.html](http://flatpack.sourceforge.net/documentation/index.html).

### Creating a Bean

**About this task**

In this section, a Java bean is created to print the id, first name, and surname from the customer information with the corresponding column names into the execution console.

**Procedure**

1. From the repository tree view, expand the **Code** node and right click the **Beans** node. In the contextual menu, select **Create Bean**. The **[New Bean]** wizard opens. In the **Name** field, type in a name for the bean, for example, **ReadOrder**. Click **Finish** to close the wizard. The bean is opened automatically in the design workspace.
2. Enter the following code in the design workspace.

```java
package beans;

import org.apache.camel.Exchange;

public class ReadOrder {

    public static void getCustomer(Exchange exch) {
        if(exch.getIn().getBody() !=null) {
            java.util.Map data = exch.getIn().getBody(java.util.Map.class);
            if(data != null) {
                String id = (String)(data.get("id"));
                System.out.println("id "+id);
                String name = (String)(data.get("name"));
                System.out.println("name "+name);
                String surname = (String)(data.get("surname"));
                System.out.println("surname "+surname);
            }
        }
    }
}
```
3. Press Ctrl+S to save your bean.

Results

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

Dropping and linking the components

About this task

![Diagram of components and connections]

Procedure

1. From the Palette, drag and drop a cFile, two cFlatPack, a cBean, and a cLog component 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

Procedure

1. Double-click Delimited_file in the design workspace to display its Basic settings view in the Component tab.
2. In the Path field, browse to or enter the file path where the flatpack file is located. In the FileName field, enter the name of the file to be processed.

3. Double-click Flat_file_parser to display its Basic settings view in the Component tab.

4. Click [...] and in the [Select a Node:] wizard, select FlatPack_config to configure the structure of the input file.

5. Double-click FlatPack_config to display its Basic settings view in the Component tab.
6. In the **PZMAP FileType** list, select **Filename** to use the PZMAP XML file from the local file system to configure the structure of the input file. Then in the **PZMAP Filename** field, browse to path where the PZMAP XML file is located.

Select the **Allow Short Lines** check box to allow lines shorter than expected by the PZMAP file. Keep the default settings of the other options.

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

8. Select the **New Instance** option to invoke a Java bean that is stored in the **Code** node of the Repository.

   In the **Bean class** field, enter the name of the bean class, `beans.ReadOrder.class` which is just created.

9. Keep the default settings of the **cLog** component to log the message exchanges.

10. Press Ctrl+S to save your Route.

**Viewing code and executing the Route**

**Procedure**

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.
As shown in the code, one route is built from "Delimited_file_cFile_1", to "cFlatPack_1". The other is built from cFlatPack_2, processed by "cBean_1" and "cLog_1" sequentially.

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 delimited file is parsed. The id, first name, and surname of the customer information with the corresponding column names is shown in the execution console.
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

<table>
<thead>
<tr>
<th>Parameters/property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters/type</td>
<td>Select the file transfer protocol, <strong>ftp</strong> or <strong>sftp</strong>, <strong>ftps</strong>.</td>
</tr>
<tr>
<td>Parameters/server</td>
<td>Type in the remote server address to be accessed.</td>
</tr>
<tr>
<td>Parameters/port</td>
<td>Type in the port number to be accessed.</td>
</tr>
<tr>
<td>Parameters/username</td>
<td>Type in the user authentication information.</td>
</tr>
<tr>
<td>Parameters/password</td>
<td>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 <strong>OK</strong> to save the settings.</td>
</tr>
<tr>
<td>Parameters/directory</td>
<td>Enter the directory you want to access on the remote server. If not specified, the root directory will be accessed.</td>
</tr>
</tbody>
</table>

Advanced settings

<table>
<thead>
<tr>
<th>Advanced</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced</td>
<td>Set the optional arguments in the corresponding table. Click [+ ] as many times as required to add arguments to the table. Then click the corresponding <strong>Value</strong> field and enter a value. See the site <a href="http://camel.apache.org/ftp.html">http://camel.apache.org/ftp.html</a> for available URI options.</td>
</tr>
</tbody>
</table>
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**

Provides HTTP-based endpoints for consuming and producing HTTP resources.

**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 su
<table>
<thead>
<tr>
<th>PUT: Request that the enclosed entity be stored under the supplied request URI.</th>
<th>bordinate of the resource identified by the request URI.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Plain Text</strong>: select the <strong>Content-Type</strong> from <strong>text/plain</strong>, <strong>text/html</strong>, <strong>text/xml</strong>, <strong>application/x-www-form-urlencoded</strong>, <strong>application/xml</strong>, <strong>application/JSON</strong>, or <strong>other</strong>... (specify the Content-Type in the next field that appears when <strong>other</strong>... is selected), and type in the text in the <strong>Content</strong> box as the request message.</td>
<td><strong>Plain Text</strong>: select the <strong>Content-Type</strong> from <strong>text/plain</strong>, <strong>text/html</strong>, <strong>text/xml</strong>, <strong>application/x-www-form-urlencoded</strong>, <strong>application/xml</strong>, <strong>application/JSON</strong>, or <strong>other</strong>... (specify the Content-Type in the next field that appears when <strong>other</strong>... is selected), and type in the text in the <strong>Content</strong> box as the request message.</td>
</tr>
<tr>
<td><strong>Form Style</strong>: click the [+ ] button to add lines as needed and define the key and value in the <strong>Parameters</strong> table. Also, enter the encoder charset in the <strong>EncoderCharset</strong> field.</td>
<td><strong>Form Style</strong>: click the [+ ] button to add lines as needed and define the key and value in the <strong>Parameters</strong> table. Also, enter the encoder charset in the <strong>EncoderCharset</strong> field.</td>
</tr>
<tr>
<td><strong>Use Message Body</strong>: use the incoming message body as the Http request. Select the <strong>Content-Type</strong> from <strong>text/plain</strong>, <strong>text/html</strong>, <strong>text/xml</strong>, <strong>application/x-www-form-urlencoded</strong>, <strong>application/xml</strong>, <strong>application/JSON</strong>, or <strong>other</strong>... (specify the Content-Type in the next field that appears when <strong>other</strong>... is selected), PUT: Request that the enclosed entity be stored under the supplied request URI.</td>
<td><strong>Use Message Body</strong>: use the incoming message body as the Http request. Select the <strong>Content-Type</strong> from <strong>text/plain</strong>, <strong>text/html</strong>, <strong>text/xml</strong>, <strong>application/x-www-form-urlencoded</strong>, <strong>application/xml</strong>, <strong>application/JSON</strong>, or <strong>other</strong>... (specify the Content-Type in the next field that appears when <strong>other</strong>... is selected), and type in the text in the <strong>Content</strong> box as the request message.</td>
</tr>
<tr>
<td><strong>Form Style</strong>: click the [+ ] button to add lines as needed and define the key and value in the <strong>Parameters</strong> table. Also, enter the encoder charset in the <strong>EncoderCharset</strong> field.</td>
<td><strong>Form Style</strong>: click the [+ ] button to add lines as needed and define the key and value in the <strong>Parameters</strong> table. Also, enter the encoder charset in the <strong>EncoderCharset</strong> field.</td>
</tr>
<tr>
<td><strong>Server Configuration / Disable Stream Cache</strong></td>
<td><strong>DefaultHttpBinding</strong> 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.</td>
</tr>
<tr>
<td><strong>Use Message Body</strong></td>
<td>use the incoming message body as the Http request. Select the <strong>Content-Type</strong> from <code>text/plain</code>, <code>text/html</code>, <code>text/xml</code>, <code>application/x-www-form-urlencoded</code>, <code>application/xml</code>, <code>application/JSON</code>, or other... (specify the Content-Type in the next field that appears when other... is selected).</td>
</tr>
<tr>
<td><strong>DELETE</strong></td>
<td>Request that the origin server delete the resource identified by the request URI. <strong>Parameters</strong>: click the <code>[+]</code> button to add lines as needed and define the key and value in the table. <strong>Encoder Charset</strong>: enter the encoder charset in the field.</td>
</tr>
<tr>
<td><strong>HEAD</strong></td>
<td>Identical to GET except that the server MUST NOT return a message body in the response. <strong>Parameters</strong>: click the <code>[+]</code> button to add lines as needed and define the key and value in the table. <strong>Encoder Charset</strong>: enter the encoder charset in the field.</td>
</tr>
<tr>
<td><strong>OPTIONS</strong></td>
<td>Represent a request for information about the communication options available on the request/response chain identified by the request URI.</td>
</tr>
<tr>
<td><strong>TRACE</strong></td>
<td>Invoke a remote, application-layer loop-back of the request message.</td>
</tr>
<tr>
<td>Server Configuration / Session Support</td>
<td>Select this check box to enable the session manager on the server side of Jetty.</td>
</tr>
<tr>
<td>Server Configuration / Use Transfer-Encoding</td>
<td>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.</td>
</tr>
<tr>
<td>Server Configuration / Enable JMX</td>
<td>Select this option to enable Jetty JMX support for this endpoint. For more information about this option, see the site <a href="http://camel.apache.org/jetty.html#Jetty-JettyJMXsupport">http://camel.apache.org/jetty.html#Jetty-JettyJMXsupport</a>.</td>
</tr>
<tr>
<td>Server Configuration / Match on URI Prefix</td>
<td>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 <a href="http://camel.apache.org/how-do-i-let-jetty-match-wildcards.html">http://camel.apache.org/how-do-i-let-jetty-match-wildcards.html</a>.</td>
</tr>
<tr>
<td>Server Configuration / Use Jetty Continuation</td>
<td>Select this check box to use Jetty continuations for the Jetty Server. For more information about this option, see the site <a href="http://wiki.eclipse.org/Jetty/Feature/Continuations">http://wiki.eclipse.org/Jetty/Feature/Continuations</a>.</td>
</tr>
<tr>
<td>Server Configuration / Enable HTTP TRACE</td>
<td>Select this option to enable the HTTP TRACE method.</td>
</tr>
</tbody>
</table>

**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. |
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

Procedure

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

Procedure

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

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

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

   ```xml
   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                    xmlns:weat="http://ws.cdyne.com/WeatherWS/"
                    xmlns:temp="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.

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.

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.

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

```java
System.out.println("-------------------RESPONSE-------------------
                    
                    
                    -------------------END-------------------");
```

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

**Viewing code and executing the Route**

**Procedure**

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

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, \texttt{cHttp} is used to as a server to start a Web service. A \texttt{cProcessor} component is used to request the service.

**Dropping and linking the components**

**Procedure**

1. Drag and drop a \texttt{cHttp} and a \texttt{cProcessor} from the Palette onto the design workspace.

![Diagram](image)

2. Link the components using a \texttt{Row > Route} connection.
3. Label the components for better identification of their roles.

**Configuring the components**

**Procedure**

1. Double-click \texttt{cHttp} to open its Basic settings view in the Component tab.

![Configuration](image)

2. In the URI field, enter \texttt{"http://localhost:8088/user"} where the service will be accessible for requests.
3. Click the Server radio button to use this \texttt{cHttp} component as a server. Keep the default settings of the other options.
4. Double-click \texttt{cProcessor} to open its Basic settings view in the Component tab.
5. In the Code area, enter the following code to get the header id of the exchange message:

```java
System.out.println(exchange.getIn().getHeader("id"));
```

6. Press Ctrl+S to save your Route.

Viewing code and executing the Route

Procedure

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

```java
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.

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Repository Type</strong></td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td><strong>Memory</strong>: messages identifiers are stored temporarily.</td>
</tr>
<tr>
<td></td>
<td><strong>File</strong>: messages identifiers are stored in a file. Specify the path to this file in the <strong>File store</strong> field.</td>
</tr>
<tr>
<td><strong>File store</strong></td>
<td>Specify the path and name of the file storing messages identifiers.</td>
</tr>
<tr>
<td><strong>Cache Size</strong></td>
<td>Type in the size of the cache, namely the number of message identifiers to store.</td>
</tr>
</tbody>
</table>
| **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**.
<table>
<thead>
<tr>
<th>Predicate</th>
<th>Type in the expression to use to specify the identifier of the messages.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Namespaces</td>
<td>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.</td>
</tr>
<tr>
<td>Eager</td>
<td>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.</td>
</tr>
<tr>
<td>SkipDuplicate</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Usage**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cIdempotentConsumer is used as a middle component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connections</td>
<td>idemp: The idemp link retrieves messages deduplicated by the cIdempotentConsumer component.</td>
</tr>
<tr>
<td></td>
<td>Route: As an optional link, the Route link retrieves all messages from the message sender.</td>
</tr>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Scenario: Deduplicating messages while routing them

This scenario applies only to a Talend solution with ESB.

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

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

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

Dropping and linking the components

**About this task**

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

**Procedure**

1. From the Palette, expand the Connectivity folder, select the cFile component, and drop it onto the design workspace as the message source component.
2. Expand the Routing folder, select the cIdempotentConsumer component and drop it onto the design workspace as the message deduplicator.
3. Expand the Custom folder, drop two cProcessor components onto the design workspace, one as the consumer for deduplicated messages and another for all messages.
4. Right-click the cFile component, select Row > Route from the contextual menu and click the cIdempotentConsumer component.
5. Right-click the cIdempotentConsumer component, select Row > idemp from the contextual menu and click the cProcessor component on the top.
6. Connect the `cIdempotentConsumer` component to the other `cProcessor` component using a **Row > Route** connection. This optional connection will retrieve all the messages coming from the source.

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

**Configuring the components and connections**

**Procedure**

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

   ![Source(cFile_1)](source_image)

   - In the **Path** field, specify the file path to the message source. From the **Encoding** list, select the encoding type of your source files, and leave all the other parameters as they are.

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

   ![Deduplicator(cIdempotentConsumer_1)](deduplicator_image)

   - From the **Repository Type** list, select between **Memory** and **File** to specify where the message identifiers will be stored before the deduplication process. For this scenario, select **File**.
   
   - In the **File store** field, specify the location of the file storing message identifiers.
   
   - In the **Expression** field, enter an expression to filter the messages. In this scenario, enter the following expression to filter the messages according to the *person* node of the XML files: `xpath("/people/person")`, and leave all the other parameters as they are. Alternatively,
you can select the **Use language** check box, select **XPath** from the **Language** list, and enter "/people/person" in the **Predicate** field.

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

![Unique(cProcessor_1)](image)

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

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

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

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

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

**Viewing code and executing the Route**

**Procedure**

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

   ```java
   public void initRoute() throws Exception {
     routeBuilder = new org.apache.camel.builder.RouteBuilder() {
       public void configure() throws Exception {
         from(uriMap.get("Source")).
           routeId("Source")
             .idempotentConsumer(
             XPath(="/people/person"),
             org.apache.camel.processor.idempotent.FileIdempotentRepository
               .fileIdempotentRepository(
                 new java.io.File("
                   /talend_files/ash/camel_temp.txt"),
                 new org.apache.camel.Processor())
               .skipDuplicate(true)
               .id("idempotentConsumer_1").process(
                 new org.apache.camel.Processor())
               }
   }
   ```

   In this partially shown piece of code, messages from the Source are filtered according to the expression `xpath("/people/person")` and deduplicated by `idempotentConsumer_1`.
2. Click the **Run** view to display it and click the **Run** button to launch the execution of your Route. You can also press **F6** to execute it.

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

### cIntercept

Intercepts each message sub-route and redirects it in another sub-route without modifying the original one. When this detour is complete, message routing to the originally intended target endpoints continues. This can be useful at testing time to simulate error handling.

### 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

<table>
<thead>
<tr>
<th>Usage rule</th>
<th><strong>cIntercept</strong> is a start component of a sub-route.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connections</strong></td>
<td><strong>Row / Route:</strong> Select the <strong>Route</strong> link to intercept all the messages of all the sub-routes listened to by the <strong>cIntercept</strong>.</td>
</tr>
<tr>
<td></td>
<td><strong>Trigger / When:</strong> Select the <strong>When</strong> link to filter the messages to intercept and click the <strong>Component</strong> view.</td>
</tr>
<tr>
<td></td>
<td>In the <strong>Type</strong> list, select the type of language you will use to declare your condition.</td>
</tr>
<tr>
<td></td>
<td>In the <strong>Condition</strong> field, type in the condition that will be used to filter the messages.</td>
</tr>
</tbody>
</table>
All the messages that do not match this condition are dropped by default or can be retrieved with the **Otherwise** link to a different channel.

| Limitation | To keep the original sub-routes untouched, **Intercept** only be used in a separate sub-route. |

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

This scenario applies only to a Talend solution with ESB.

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

![Diagram of scenario](image)

Dropping and linking the components

**About this task**

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

**Procedure**

1. From the **Connectivity** folder of the **Palette**, drop four **cFile** components onto the design workspace.
2. Connect the two pairs of **cFile** components using **Row > Route** connections. Messages on these two sub-routes will be intercepted.
3. From the **Exception Handling** folder, drop a **cIntercept** component onto the design workspace.
4. From the **Custom** folder, drop a **cProcessor** component onto the design workspace.
5. From the **Connectivity** folder, drop a fifth **cFile** component onto the design workspace.
6. From the **Miscellaneous** folder, drop a **cStop** component onto the design workspace.
7. Connect these four components one to the next using **Row > Route** connections. Along this sub-route, intercepted messages will be directed to a new endpoint before the entire Route is terminated.
8. Label the components to better identify their roles in the Route.
Configuring the components and connections

About this task

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

Procedure

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

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

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

4. In the Path field, specify the file path to the first destination you are going to send messages to, and leave the other parameters as they are.
5. In the same way, set the cFile components labeled Sender_2 and Receiver_2 across the second sub-route.

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

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

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

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

9. Press Ctrl+S to save your Route.

**Viewing code and executing the Route**

**Procedure**

1. Click the Code tab at the bottom of the design workspace to have a look at the generated code.
As shown in this piece of code, Interceptor intercepts all messages on route, the intercepted messages are directed .to the endpoint Receiver_3, and cStop_1 terminates message routing before the messages are routed from the endpoint Sender_1 .to the endpoint Receiver_1 and from the endpoint Sender_2 .to the endpoint Receiver_2.

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

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

Implements producers and consumers of message exchanges or implements a message translator using the Java Domain Specific Language (DSL).

cJavaDSLProcessor can quickly whirls 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 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 346.

cJMS

Exchanges messages between a Route and a JMS provider.

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 on page 246.

Limitation

n/a

Scenario 1: Sending and receiving a message from a JMS queue

This scenario applies only to a Talend solution with ESB.

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

Procedure

1. From the Palette, expand the Connectivity/Messaging folder, and drop a cMQConnectionFactory component onto the design workspace to specify the JMS connection factory for handling messages.
2. From the Connectivity folder, drop one cFile and two cJMS components onto the design workspace.
3. From the Custom folder, drop a cProcessor component onto the design workspace.
4. Connect the cFile component to a cJMS component using a Row > Route connection as the message producer sub-route.
5. Connect the other cJMS component to the cProcessor component using a Row > Route connection as the message consumer sub-route.
6. Label the components properly for better identification of their functionalities.

Configuring the components

Procedure

1. Double-click the cMQConnectionFactory component to display its Basic settings view in the Component tab.
2. From the MQ Server list, select an MQ server. In this use case, we use the default ActiveMQ server to handle the messages.
In the **Broker URI** field, type in the URI of the message broker. Here we simply use the default URI "vm://localhost?broker.persistent=false".

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

![File_source(cFile_1)](image)

4. Define the properties of the **cFile** component.
   In this use case, simply specify the path to the folder that holds the source file to be sent as electronic message, and leave the other parameters as they are.

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

![Message_producer(cJMS_1)](image)

6. From the **Type** list, select **queue** to send the messages to a JMS queue.
   In the **Destination** field, type in a name for the JMS queue, "queue.hello" in this use case. Double-click the [...] button next to **ConnectionFactory**. Select the MQ connection factory that you have just configured in the dialog box and click **OK**.

![Select a Node:](image)

7. Switch to the message consumer sub-route, and double click the **cJMS** component labeled **Message_consumer** to display its **Basic settings** view.
8. Configure the message consumer using exactly the same parameters as in the message producer.

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

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

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

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

### Viewing code and executing the Route

**Procedure**

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

    ```java
    public void configure() throws java.lang.Exception {
        from(uriMap.get("cFile_1")).routeId("File_source_cFile_1")
            .to(uriMap.get("cJMS_1")).id("cJMS_1");
        from(uriMap.get("cJMS_2")).routeId("Message_consumer_cJMS_2")
            .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");
    }
    ```
In the partially shown code, a message route is built from the `File_source_cFile_1` to the `cJMS_1` which then sends the message to a message queue via a broker identified by `vm://localhost?broker.persistent=false`. The message from the `Message_consumer_cJMS_2` is processed by `cProcessor_1`.

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

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

**Scenario 2: Setting up a JMS local transaction**

This scenario applies only to a Talend solution with ESB.

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

1. The first Route is used to send a "hello world!" message to feed the `queue.hello` JMS queue.
2. The second Route is used to test the received JMS message. This message is redelivered six times to the `queue.hello` queue and is then moved to the Dead Letter JMS queue. The Route is programmed to throw an exception every time an exchange is processed by the Route.
3. The last Route is used to consume the "hello world!" message from the Dead Letter JMS queue.

In this use case, we use the local ActiveMQ server to handle messages between different Routes. The ActiveMQ server need to be started before executing the Routes. For more information on installing and launching ActiveMQ server, see the Apache Web site [http://activemq.apache.org/index.html](http://activemq.apache.org/index.html).
Sending a message to the `queue.hello` JMS queue

Dropping and linking the components

Procedure

1. From the Palette, drop the five following components onto the design workspace: one `cMQConnectionFactory`, one `cConfig`, one `cMessagingEndpoint`, one `cJMS` and one `cProcessor` component.
2. Connect the `cMessagingEndpoint` component to the `cJMS` using a Row > Route connection.
3. Connect the `cJMS` component to the `cProcessor` component using a Row > Route connection.

Configuring the components

Procedure

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

2. From the MQ Server list, select an MQ server. In this use case, we use the default ActiveMQ server to handle the messages.
3. In the Broker URI field, type in ActiveMQ's default URI of the localhost server: "tcp://localhost:61616".
4. Double-click the `cConfig` component, which is labelled `DatasetConfig`, to display its Basic settings view in the Component tab and set its parameters.
5. Write a piece of code in the Code field to register the dataset instance `hello` into the registry, as shown below.

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

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

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

8. Double-click the cJMS component labeled `AMQ_Send` to display its Basic settings view.
9. From the **Type** list, select **queue** to send the message to a JMS queue.
   In the **Destination** field, type in a name for the JMS queue, "queue.hello" in this use case.
   Double-click the [...] button next to **ConnectionFactory**. Select the MQ connection factory that you have just configured in the dialog box and click **OK**.

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

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

**Executing the Route**

**Procedure**

Click the **Run** button in the **Run** view to launch the execution of your Route. You can also press **F6** to execute it.
RESULT: One "hello world!" message is sent to the JMS Queue, as shown in the **Run** console.
Testing the received message

Linking the components

Procedure

1. From the Palette, drop the four following components onto the design workspace: one cJMS, two cProcessor components and one cMQConnectionFactory.
2. Connect the cJMS component to the first cProcessor using a Row > Route connection.
3. Connect the first cProcessor component to the second cProcessor component using a Row > Route connection.

Defining the settings of the components

Procedure

1. Double-click the cMQConnectionFactory component labelled AMQ_Rev_ConnectionFactory to display its Basic settings view in the Component tab.
2. From the **MQ Server** list, select an MQ server. In this use case, we use the default ActiveMQ server to handle the messages.

   Select the **Use transaction** check box.

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

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

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

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

   Double-click the [...] button next to **ConnectionFactory**. Select the MQ connection factory that you have just configured in the dialog box and click **OK**.

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

   ```java
   System.out.println("AMQ Receive: "+
   exchange.getIn().getBody(String.class));
   ```
7. Double-click the second cProcessor component labelled *ThrowEx* to display its **Basic settings** view in the **Component** tab, and customize the code in the **Code** area to throw the *Force fail* exception every time an exchange is processed by the route.

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

**Running the Route**

**Procedure**

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

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

**Consuming the message from the *DeadLetter JMS* queue**

![Diagram of message flow]

**Arranging the flow of the message**

**Procedure**

1. From the **Palette**, drop the three following components onto the design workspace: one cMQConnectionFactory, one cJMS and one cProcessor component.
2. Connect the cJMS component to the cProcessor component using a **Row > Route** connection.

**Configuring how the message is processed**

**Procedure**

1. Double-click the cMQConnectionFactory component to display its **Basic settings** view in the **Component** tab.
2. From the **MQ Server** list, select an MQ server. In this use case, we use the default ActiveMQ server to handle the messages.

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

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

5. From the **Type** list, select **queue** to send the messages to a JMS queue.
   
   In the **Destination** field, type in a name for the JMS queue, "ActiveMQ.DLQ" in this use case (the default Dead Letter Queue in ActiveMQ).

   Double-click the [...] button next to **ConnectionFactory**. Select the MQ connection factory that you have just configured in the dialog box and click **OK**.

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

   ```java
   System.out.println("AMQ Receive: "+
   exchange.getIn().getBody(String.class));
   ```
Launching the execution of the Route

Procedure

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

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

Scenario 3: Sending and receiving a scheduled delivery of messages from a JMS Queue using Camel Quartz

This scenario applies only to a Talend solution with ESB.

This scenario will show you how to use the Camel Quartz component to provide a scheduled delivery of messages from a JMS Queue.

To do this, we will build two Routes, a message producer Route and a consumer Route. We will implement the Quartz component in the producer Route to send scheduled messages to a JMS Queue. The messages are then consumed by the consumer Route.

In this use case, we will use Apache ActiveMQ as the message broker. We 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

About this task

![Diagram showing the components and their connections.]

Procedure

1. From the Palette, drag and drop a `cMQConnectionFactory`, a `cJMS`, a `cSetBody`, and two `cMessagingEndpoint` 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

Procedure

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

![Basic settings view of cMQConnectionFactory.]

2. From the MQ Server list, select ActiveMQ to handle messages.
   
   In the Broker URI field, type in the URI of the local ActiveMQ server, "tcp://localhost:61616".
   
   Select the Use PooledConnectionFactory check box and keep the default settings.
3. Double-click the **quartzConsumer** component to open its **Basic settings** view in the **Component** tab.

4. In the **URI** field, enter the code "quartz://HelloWorld?trigger.repeatInterval=2000&trigger.repeatCount=-1" to define a timer for starting message exchanges. In this use case, we want the message to be delivered endlessly between an interval of two seconds. For more information about Quartz, see the site [http://camel.apache.org/quartz.html](http://camel.apache.org/quartz.html).

5. Click the **Advanced settings** view. Click + at the bottom of the **Dependencies** list to add a row and select **quartz** from the drop-down list. For more information about the Quartz component, see the site [http://camel.apache.org/quartz.html](http://camel.apache.org/quartz.html).

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

7. Select **Simple** from the **Language** list box and type in "Hello world" in the **Expression** field.

8. Double-click the **logMessage** component to open its **Basic settings** view in the **Component** tab.
9. In the URI field, enter "log:quartzMessage" where the message exchanges are logged.

10. Double-click the jmsProducer component to display its Basic settings view in the Component tab.

11. From the Type list, select queue to send the messages to a JMS queue.
    In the Destination field, type in a name for the JMS queue, "quartzTest" in this use case.
    In the ConnectionFactory field, click [...] and select the MQ connection factory that you have just configured.

12. Press Ctrl+S to save your Route.

**Viewing the code and executing the Route**

**Procedure**

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("quartzConsumer")).routeId("quartzConsumer").setBody().simple("Hello world").id("cSetBody_1").to{
                uriMap.get("logMessage").id("cMessagingEndpoint_2").to{
                    uriMap.get("jmsProducer").id("cJMS_1");
            }
        }
    }
    getCamelContext().get(0).addRoutes(routeBuilder);
}
```

As shown above, the message flow from quartzConsumer is given a payload by cSetBody_1 and then sent to logMessage and jmsProducer.
2. Press F6 to execute the Route.
RESULT: The logs of the message exchange are printed in the console.

Building the consumer Route

Arranging the flow of the message

About this task

Procedure

1. From the Palette, drag and drop a cMQConnectionFactory, a cJMS, and a cMessagingEndpoint 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

Procedure

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

   ![cMQConnectionFactory Component](image)

   - **MQ Server**: Select `ActiveMQ`
   - **Broker URI**: Enter `tcp://localhost:61616`
   - **Max Connections**: 1
   - **Max Active**: 500
   - **Idle Timeout**: 30000
   - **Expiry Timeout**: 60000

2. Configure the `cMQConnectionFactory` component the same as in the producer Route.
3. Double-click the `jmsConsumer` component to display its **Basic settings** view in the **Component** tab.
4. Configure the `jmsConsumer` component the same as the `jmsProducer` component in the producer Route to consume the messages in the defined queue "quartzTest".
5. Double-click the `logMessage` component to open its **Basic settings** view in the **Component** tab.

   ![logMessage Component](image)

   - **URI**: Enter "log:quartzMessage"

6. In the **URI** field, enter "log:quartzMessage" where the message exchanges are logged.
7. Press **Ctrl+S** to save your Route.

Executing the Route

Procedure

1. Click the **Code** tab at the bottom of the design workspace to check the generated code.
As shown above, the message flow is routed from `jmsConsumer` to `logMessage`.

2. Press F6 to execute the Route.

RESULT: The logs of the message exchange are printed in the console.

**cKafka**

Communicates with 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**

<table>
<thead>
<tr>
<th>Broker List</th>
<th>Specify the list of Kafka message brokers in the form hostname1:port1, hostname2:port2, hostname3:port3.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client Id</strong></td>
<td>Specify an Id string of the client to pass to the server when making requests.</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Topic</strong></td>
<td>Type in a name for the message topic in the message broker.</td>
</tr>
<tr>
<td><strong>GroupId</strong></td>
<td>Enter the Id of the Kafka Connect cluster group.</td>
</tr>
<tr>
<td><strong>Partitioner</strong></td>
<td>Enter the partitioner that determines how data is distributed across the Kafka cluster.</td>
</tr>
<tr>
<td><strong>Serializer Class</strong></td>
<td>Enter the class name of the serializer to be used.</td>
</tr>
<tr>
<td><strong>Key Serializer Class</strong></td>
<td>Enter the class name of the key serializer to be used.</td>
</tr>
<tr>
<td><strong>Send Buffer (bytes)</strong></td>
<td>The size of the TCP send buffer to use when sending data.</td>
</tr>
<tr>
<td><strong>Request Required Acks</strong></td>
<td>Specify whether the producer waits for a acknowledgement from the broker that the message was received by entering:</td>
</tr>
<tr>
<td> </td>
<td>• <strong>0</strong>, which means that the producer never waits for an acknowledgement from the broker;</td>
</tr>
<tr>
<td> </td>
<td>• <strong>1</strong>, which means that the producer gets an acknowledgement after the leader replica has received the data;</td>
</tr>
<tr>
<td> </td>
<td>• <strong>-1</strong> or <strong>all</strong>, which means that the producer gets an acknowledgement after all in-sync replicas have received the data.</td>
</tr>
<tr>
<td><strong>Request Timeout (ms)</strong></td>
<td>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</td>
</tr>
</tbody>
</table>
| **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...
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Max Block (ms)</strong></td>
<td>Specify the maximum amount of time in milliseconds that the producer will wait either because the buffer is full or metadata unavailable.</td>
</tr>
<tr>
<td><strong>Max Request Size</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Receive Buffer (bytes)</strong></td>
<td>Specify the size of the TCP receive buffer to use when reading data. If the value is −1, the OS default will be used.</td>
</tr>
<tr>
<td><strong>Max in Flight Request</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Metadata Max Age (ms)</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Reconnect Backoff (ms)</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

The following options are available only when the **cKafka** is used as a Consumer:
<table>
<thead>
<tr>
<th>Send Buffer (bytes)</th>
<th>Specify the size of the TCP send buffer to use when sending data. If the value is <code>-1</code>, the OS default will be used.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retry Backoff (ms)</td>
<td>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.</td>
</tr>
<tr>
<td>Connection Idle Max (ms)</td>
<td>Specify the time in milliseconds after which idle connections will be closed.</td>
</tr>
<tr>
<td>Receive Buffer (bytes)</td>
<td>Specify the size of the TCP receive buffer to use when reading data. If the value is <code>-1</code>, the OS default will be used.</td>
</tr>
<tr>
<td>Metadata Max Age (ms)</td>
<td>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.</td>
</tr>
<tr>
<td>Reconnect Backoff (ms)</td>
<td>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.</td>
</tr>
<tr>
<td>Barrier Await Timeout (ms)</td>
<td>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.</td>
</tr>
<tr>
<td>Auto Commit Enable</td>
<td>Select this check box to periodically commit the offset of messages in the background. Specify the frequency in milliseconds in the <strong>Auto Commit Interval (ms)</strong> that the consumer offsets are committed to Kafka.</td>
</tr>
<tr>
<td><strong>Fetch Min (bytes)</strong></td>
<td>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.</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Fetch Wait Max (ms)</strong></td>
<td>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 <strong>Fetch Min (bytes)</strong>.</td>
</tr>
</tbody>
</table>
| **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.

<table>
<thead>
<tr>
<th>Maximum Partition Fetch (bytes)</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session Timeout (ms)</td>
<td>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.</td>
</tr>
<tr>
<td>Partition Assignor</td>
<td>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.</td>
</tr>
<tr>
<td>Request Timeout (ms)</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 site [http://kafka.apache.org/documentation.html](http://kafka.apache.org/documentation.html) for available options.

<table>
<thead>
<tr>
<th><strong>SSL Key Password</strong></th>
<th>Click [...] and enter the password of the private key in the key store file in double quotes.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSL Keystore Location</strong></td>
<td>Enter the location of the key store file.</td>
</tr>
<tr>
<td><strong>SSL Keystore Password</strong></td>
<td>Click [...] and enter the password for the key store file in double quotes. This is only needed if <strong>SSL Keystore Location</strong> is configured.</td>
</tr>
<tr>
<td><strong>SSL Truststore Location</strong></td>
<td>Enter the location of the trust store file.</td>
</tr>
<tr>
<td><strong>SSL Truststore Password</strong></td>
<td>Click [...] and enter the password for the trust store file in double quotes.</td>
</tr>
<tr>
<td><strong>SSL Cipher Suits</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>SSL Endpoint Algorithm</strong></td>
<td>Enter the endpoint identification algorithm to validate server hostname using server certificate.</td>
</tr>
<tr>
<td><strong>Kerberos Service Name</strong></td>
<td>Enter the Kerberos principal name that Kafka runs as.</td>
</tr>
<tr>
<td><strong>Security Protocol</strong></td>
<td>Select the protocol to use to communicate with brokers from <strong>Plaintext</strong>, <strong>SSL</strong>, <strong>SASL over Plaintext</strong>, and <strong>SASL over SSL</strong>.</td>
</tr>
</tbody>
</table>
Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cKafka is used as a start, middle or end component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Related scenarios

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

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

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Select between Random, Round Robin, Sticky, Topic, Failover, and Custom. Each method is described below.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>The receiving endpoint is chosen randomly at each exchange.</td>
</tr>
<tr>
<td>Round Robin</td>
<td>Messages are distributed according to the round robin method which distributes the load evenly.</td>
</tr>
<tr>
<td>Sticky</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Expression: Type in the expression that will be used to calculate a correlation key that will determine the endpoint to choose.</td>
</tr>
<tr>
<td>Topic</td>
<td>Select this option to send all the messages to all the endpoints.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Failover</td>
<td><strong>Basic mode:</strong> 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.&lt;br&gt;&lt;br&gt;<strong>Specify exceptions:</strong> Specify the exceptions to which the failover should react to in the Exception table.&lt;br&gt;&lt;br&gt;<strong>Use with Round robin:</strong> Select this option to use failover with advanced options.&lt;br&gt;&lt;br&gt;From the <strong>Maximum failover attempt</strong> list, select the number of attempt to be proceed before giving up the transfer:&lt;br&gt;- <strong>Attempt forever:</strong> always attempts to transfer the messages and always try to failover.&lt;br&gt;- <strong>Never failover:</strong> gives up immediately the transfer of messages and never try to failover.&lt;br&gt;- <strong>A number of attempts:</strong> attempts $n$ number of time to transfer messages, specify that number in the <strong>Number of attempts</strong> field.&lt;br&gt;&lt;br&gt;<strong>Inherit error handler:</strong> 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.&lt;br&gt;&lt;br&gt;<strong>Use Round robin:</strong> 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.</td>
</tr>
</tbody>
</table>
## Custom Load balancer

Type in the name of your custom load balancer.

### Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cLoadBalancer is used as a middle component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connections</td>
<td>Load Balance: Select this link to route messages to the next endpoint according to the selected load-balancing strategy.</td>
</tr>
<tr>
<td></td>
<td>Route: Select this link to route all the messages from the sender to the next endpoint.</td>
</tr>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

### Scenario: Distributing messages to receiver endpoints based on round robin

This scenario applies only to a Talend solution with ESB.

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

### Dropping and linking the components

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

1. From the Connectivity folder of the Palette, drop one cFile component and two cMessagingEndpoint components onto the workspace, and label them according to their roles in the Route: Sender, Receiver_A, and Receiver_B respectively.
2. From the Routing folder, drop a cLoadBalancer component onto the design workspace, and label it Load_balancer.
3. From the Core folder, drop two cJavaDSLProcessor components onto the design workspace, and label them according to their roles in the Route: To_Receiver_A and To_Receiver_B respectively.
4. From the Custom folder, drop two cProcessor components onto the design workspace, and label them according to their roles in the Route: Monitor_A, and Monitor_B respectively.
5. Link the cFile component to the cLoadBalancer component using a Row > Route connection.
6. Link cLoadBalancer to each of the two cJavaDSLProcessor components using a Row > Load Balance connection.
7. Link each of the two cMessagingEndpoint components to the corresponding cProcessor component using a Row > Route connection.

Configuring the components and connections

Procedure

1. Double-click the cFile component to open its Basic Settings view in the Component tab.
2. In the **Path** field, specify the file path to message source.

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

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

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

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

![cMessagingEndpoint component labeled Receiver_A](image)

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

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

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

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

8. Press Ctrl+S to save your Route.

**Viewing the code and executing the Route**

**Procedure**

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

   ```java
   public void initRoute() throws Exception {
     routeBuilder = new org.apache.camel.builder.RouteBuilder() {
       public void configure() throws Exception {
         from(uriMap.get("Sender")).routeId("Sender")
           .loadBalance().roundRobin().id("cLoadBalancer_1")
           .to("direct:a").id("cJavaDSLProcessor_1")
           .to("direct:b").id("cJavaDSLProcessor_2");
     }
   }
   
   As shown above, while messages are routed from the source endpoint `Sender` to the destination endpoints, routing load balancing is implemented according to the `roundRobin()` method by `cLoadBalancer_1`.

2. Press F6 to run your Route.

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

Logs message exchanges in a Route.

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

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logging Category</strong></td>
<td>Enter the name of the logging category to use.</td>
</tr>
<tr>
<td><strong>Level</strong></td>
<td>Select a logging level from DEBUG, ERROR, INFO, OFF, TRACE, or WARN.</td>
</tr>
<tr>
<td><strong>Use default output log message</strong></td>
<td>Select this option to use the default output log message provided by the underlying logging mechanism.</td>
</tr>
<tr>
<td><strong>Options / None (For default output log message only)</strong></td>
<td>Select this option to take no action on the log message.</td>
</tr>
</tbody>
</table>
| **Options / Specifies a group size for throughput logging (For default output log message only)** | Select this option to use throughput logging and specify a group size for the throughput logging.  
  **Size**: Enter an integer that specifies a group size for throughput logging. |
| **Options / Group message stats by time interval (in millis) (For default output log message only)** | Select this option to use throughput logging and group message statistics.  
  **Interval**: Specify the time interval (in milliseconds) by which the message statistics will be grouped.  
  **Delay**: Set the initial delay (in milliseconds) for message statistics. |
Options / Format the log output (For default output log message only)

Select this option to 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 219.

cLoop

Processes messages repetitively 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

<table>
<thead>
<tr>
<th>Loop Type</th>
<th>Select a type of loop to be carried out: Expression, Header, or Value.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Expression: Use an expression to determine the loop count.</td>
</tr>
<tr>
<td></td>
<td>Header: Use a header to determine the loop count.</td>
</tr>
<tr>
<td></td>
<td>Value: Use an argument to set the loop count.</td>
</tr>
<tr>
<td></td>
<td>When using Expression: In the Language field, select the language of the expression you want to use to determine the loop count between</td>
</tr>
</tbody>
</table>
Constant, EL, Groovy, Header, Javascript, JoSQL, JXPath, MVEL, None, OGNL, PHP, Property, Python, Ruby, Simple, SpEL, SQL, XPath, XQuery. Type 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**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th><strong>cLoop</strong> can be a middle component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Limitation</strong></td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Related scenarios**

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

Sends or receives 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

<table>
<thead>
<tr>
<th>Protocols</th>
<th>List of protocols for sending or receiving mails.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Host name of the mail server.</td>
</tr>
<tr>
<td>Port</td>
<td>Port number of the mail server.</td>
</tr>
<tr>
<td>UserName and Password</td>
<td>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.</td>
</tr>
<tr>
<td>Subject</td>
<td>Subject of the mail being sent.</td>
</tr>
<tr>
<td>Content Type</td>
<td>The mail content type.</td>
</tr>
<tr>
<td>From</td>
<td>The mail sender.</td>
</tr>
<tr>
<td>To</td>
<td>The mail receivers.</td>
</tr>
<tr>
<td>CC</td>
<td>The CC recipients of the mail. Separate multiple email addresses with a comma.</td>
</tr>
<tr>
<td>BCC</td>
<td>The BCC recipients of the mail. Separate multiple email addresses with a comma.</td>
</tr>
</tbody>
</table>
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](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

Procedure

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.

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.

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.
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

#### Procedure

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.

<table>
<thead>
<tr>
<th>Basic settings</th>
<th>Protocols</th>
<th>imaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced settings</td>
<td>Host</td>
<td>imap.gmail.com</td>
</tr>
<tr>
<td>Dynamic settings</td>
<td>Port</td>
<td>993</td>
</tr>
<tr>
<td>View</td>
<td>UserName</td>
<td>context.USERNAME</td>
</tr>
<tr>
<td>Documentation</td>
<td>Subject</td>
<td>Content-Type: text/plain</td>
</tr>
<tr>
<td></td>
<td>From</td>
<td>To</td>
</tr>
<tr>
<td></td>
<td>CC</td>
<td>BCC</td>
</tr>
</tbody>
</table>

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.

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

    ```java
    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

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.


#### 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**

|-----------------|----------------------------------------------------------------------------------------------------------|


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: Select this link to route the filtered messages to the next endpoint.

Route: Select this link to route all the messages from the sender to the next endpoint.

Limitation

n/a

Scenario: Filtering messages according to a criterion

This scenario applies only to a Talend solution with ESB.

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

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

```
<person>
  <firstName>Pierre</firstName>
  <lastName>Dupont</lastName>
  <city>Paris</city>
</person>
```
Dropping and linking the components

About this task

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

![Diagram of component connections]

Procedure

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

Configuring the components and connections

Procedure

1. Double-click the `cFile` component labeled `Sender` to open its Basic settings view in the Component tab.
2. In the Path field, specify the file path to message source.

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

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

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

   In this scenario, we want to sort out the XML files containing a city node with the value of Paris, so we select XPath from the Language list, and fill the in the Expression field with this expression: 

   
   
   

   
   


6. Double-click the cFile component labeled Paris_only to open its Basic settings view in the Component view, and specify the path for the messages meeting the filter criterion in the Path field.
Repeat this step to define the path for all the messages from the sender in the cFile component labeled Unfiltered.

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

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

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

8. Press Ctrl+S to save your Route.

**Viewing the code and executing the Route**

**Procedure**

1. To have a look at the generated code, click the Code tab at the bottom of the design workspace.
As shown in this piece of code, messages from the sender are filtered by `cMessageFilter_1` according to `.xpath("/person[city='Paris']")` and the messages matching the filter are sent to the endpoint `Paris_only`, while all messages are sent to the endpoint `Unfiltered`.

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

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

```
cMessageRouter
  Creates different channels for each filtered message type according to specified conditions so that messages can later on be treated more accurately in each new channel.
```
**Usage**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connections</td>
<td>Row / Route: Select this link to pass on all the messages to the next component.</td>
</tr>
<tr>
<td></td>
<td>Trigger / When: Select the When link and click the Component view.</td>
</tr>
<tr>
<td></td>
<td>In the Type list, select the type of language you will use to declare your condition from none, bean, constant, correlation, el, groovy, header, javascript, sql, jsonpath, xpath, mvel, ognl, php, property, python, ruby, simple, spel, sql, xpath, and xquery.</td>
</tr>
<tr>
<td></td>
<td>Append endChoice(): If the When link is followed by a cMessageFilter or cLoadBalancer and so on, there will be a compile error. This is because once there is such a component there, the flow type will change from choice() to loadbalance(), which can not be followed by a When or Otherwise link. Select this check box to change the flow type back from loadbalance() to choice().</td>
</tr>
<tr>
<td></td>
<td>In the Condition field, type in the condition that will be used to filter the messages.</td>
</tr>
<tr>
<td></td>
<td>All the messages that do not match this condition are retrieved with the Otherwise link to a</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>different channel or dropped if an <strong>Otherwise</strong> link does not present.</td>
<td>Trigger / Otherwise: This link automatically retrieves the messages that do not match the <strong>When</strong> conditions.</td>
</tr>
<tr>
<td><strong>Note:</strong> There can be more than one <strong>When</strong> link in a Route.</td>
<td><strong>Note:</strong> There can be only one <strong>Otherwise</strong> link, which is optional, in a Route.</td>
</tr>
</tbody>
</table>

**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`.

![Diagram of routing messages](image-url)
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:

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

Dropping and linking the components

Procedure

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

About this task

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).

Procedure

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

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.

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.

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.

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**

**Procedure**

1. Click the Code tab at the bottom of the design workspace to have a look at the generated code.
As shown in the code, the messages are routed according to conditions initialized with the \texttt{.choice()} piece of code. The filter you defined is initialized with the \texttt{.when()} piece of code, and the non filtered messages are routed through the \texttt{.otherwise()} piece of code.

2. Click the \textbf{Run} button in the \textbf{Run} view or press \textbf{F6} to execute your Route.

RESULT: The files containing "Paris" are sent to a folder named \textit{Paris\_only}, and the other messages are sent in a folder called \textit{Other\_cities}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.png}
\caption{Screenshot of the Job \texttt{cMessageRouter} execution.}
\end{figure}

\textbf{cMessagingEndpoint}

Allows two applications to communicate by either sending or receiving messages.
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.

<table>
<thead>
<tr>
<th>Component / ArtifactId / URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHC (camel-ahc) ahc:httpUri</td>
<td>To call external HTTP services using Async Http Client.</td>
</tr>
<tr>
<td>AHC Websocket (camel-ahc-ws) ahc-ws:httpUri</td>
<td>To exchange data with external Websocket servers using Async Http Client.</td>
</tr>
<tr>
<td>APNS (camel-apns) apns:name</td>
<td>For sending notifications to Apple iOS devices.</td>
</tr>
<tr>
<td>Avro (camel-avro) avro:transport:host:port/mess ageName</td>
<td>Working with Apache Avro for data serialization.</td>
</tr>
<tr>
<td>Atom (camel-atom) atom:feedUri</td>
<td>For consuming Atom RSS feeds.</td>
</tr>
<tr>
<td>Braintree (camel-braintree) braintree:apiName/methodName</td>
<td>For integrating with the Braintree Payment System.</td>
</tr>
<tr>
<td>Camel Context (camel-context) context:contextId:localEndpointUrl</td>
<td>To send/receive messages between Camel routes in a black box way. This component is deprecated.</td>
</tr>
<tr>
<td>CMIS (camel-cmis) cmis:cmsUrl</td>
<td>The cmis component uses the Apache Chemistry client API and allows you to add/ read nodes to/from a CMIS compliant content repositories.</td>
</tr>
<tr>
<td>CoAP (camel-coap) coap:uri</td>
<td>For sending and receiving messages from COAP capable devices.</td>
</tr>
<tr>
<td>CouchDB (camel-couchdb) couchdb:protocol:hostname:port/database</td>
<td>To integrate with CouchDB databases.</td>
</tr>
<tr>
<td>Component / ArtifactId / URI</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Crypto (JCE) (camel-crypto)</strong>&lt;br&gt;crypto:cryptoOperation:name</td>
<td>For signing and verifying exchanges using the Signature Service of the Java Cryptographic Extension (JCE).</td>
</tr>
<tr>
<td><strong>CXF (camel-cxf)</strong>&lt;br&gt;cxf:beanId:address</td>
<td>Works for SOAP WebServices using Apache CXF.</td>
</tr>
<tr>
<td><strong>Disruptor (camel-disruptor)</strong>&lt;br&gt;disruptor:name</td>
<td>To provide asynchronous SEDA behavior using LMAX Disruptor.</td>
</tr>
<tr>
<td><strong>EHCache (camel-cache)</strong>&lt;br&gt;cache:cacheName</td>
<td>To perform caching operations using EHCache as the Cache Implementation. This component is deprecated.</td>
</tr>
<tr>
<td><strong>Elasticsearch (camel-elasticsearch)</strong>&lt;br&gt;elasticsearch:clusterName</td>
<td>For interfacing with ElasticSearch server.</td>
</tr>
<tr>
<td><strong>EISSL (camel-elsql)</strong>&lt;br&gt;elsql:elsqlName:resourceUri</td>
<td>The EISSL component is an extension to the existing SQL Component that uses EISSL to define the SQL queries.</td>
</tr>
<tr>
<td><strong>etcd (camel-etcd)</strong>&lt;br&gt;etcd:namespace/path</td>
<td>To work with Etcd, a distributed reliable key-value store.</td>
</tr>
<tr>
<td><strong>Exec (camel-exec)</strong>&lt;br&gt;exec:executable</td>
<td>To execute OS system commands.</td>
</tr>
<tr>
<td><strong>Facebook (camel-facebook)</strong>&lt;br&gt;facebook:methodName</td>
<td>To provide access to all of the Facebook APIs accessible using Facebook4J.</td>
</tr>
<tr>
<td><strong>Flatpack (camel-flatpack)</strong>&lt;br&gt;flatpack:type:resourceUri</td>
<td>The flatpack component supports fixed width and delimited file parsing via the FlatPack library.</td>
</tr>
<tr>
<td><strong>FOP (camel-fop)</strong>&lt;br&gt;fop:outputType</td>
<td>To render a message into different output formats using Apache FOP.</td>
</tr>
<tr>
<td><strong>Freemarker (camel-freemarker)</strong>&lt;br&gt;freemarker:resourceUri</td>
<td>Transforms the message using a FreeMarker template.</td>
</tr>
<tr>
<td><strong>FTP (camel-ftp)</strong>&lt;br&gt;ftp:host:port/directoryName</td>
<td>For uploading or downloading files from FTP servers.</td>
</tr>
<tr>
<td><strong>Geocoder (camel-geocoder)</strong>&lt;br&gt;geocoder:address:latlng</td>
<td>For looking up geocodes (latitude and longitude) for a given address or reverse lookup.</td>
</tr>
<tr>
<td>Component / ArtifactId / URI</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Git</strong> (camel-git)</td>
<td>For working with git repositories.</td>
</tr>
<tr>
<td>git:localPath</td>
<td></td>
</tr>
<tr>
<td><strong>Guava EventBus</strong> (camel-guava-eventbus)</td>
<td>Provides integration bridge between Camel and Google Guava EventBus.</td>
</tr>
<tr>
<td>guava-eventbus:eventBusRef</td>
<td></td>
</tr>
<tr>
<td><strong>Grape</strong> (camel-grape)</td>
<td>To fetch, load and manage additional jars when CamelContext is running.</td>
</tr>
<tr>
<td>grape:defaultCoordinates</td>
<td></td>
</tr>
<tr>
<td><strong>HBase</strong> (camel-hbase)</td>
<td>For reading/writing from/to an HBase store (Hadoop database).</td>
</tr>
<tr>
<td>hbase:tableName</td>
<td></td>
</tr>
<tr>
<td><strong>HDFS</strong> (camel-hdfs)</td>
<td>For reading/writing from/to an HDFS file system using Hadoop 1.x. This component is deprecated.</td>
</tr>
<tr>
<td>hdfs:hostName:port/path</td>
<td></td>
</tr>
<tr>
<td><strong>HDFS2</strong> (camel-hdfs2)</td>
<td>For reading/writing from/to an HDFS file system using Hadoop 2.x.</td>
</tr>
<tr>
<td>hdfs2:hostName:port/path</td>
<td></td>
</tr>
<tr>
<td><strong>HTTP4</strong> (camel-http4)</td>
<td>For calling out to external HTTP servers using Apache HTTP Client 4.x.</td>
</tr>
<tr>
<td>http4:httpUri</td>
<td></td>
</tr>
<tr>
<td><strong>Ignite Cache</strong> (camel-ignite)</td>
<td>The Ignite Cache endpoint is one of camel-ignite endpoints which allows you to interact with an Ignite Cache.</td>
</tr>
<tr>
<td>ignite-cache:cacheName</td>
<td></td>
</tr>
<tr>
<td><strong>Infinispan</strong> (camel-infinispan)</td>
<td>For reading/writing from/to Infinispan distributed key/value store and data grid.</td>
</tr>
<tr>
<td>infinispan:cacheName</td>
<td></td>
</tr>
<tr>
<td><strong>IronMQ</strong> (camel-ironmq)</td>
<td>The ironmq provides integration with IronMQ an elastic and durable hosted message queue as a service.</td>
</tr>
<tr>
<td>ironmq:queueName</td>
<td></td>
</tr>
<tr>
<td><strong>JBPM</strong> (camel-jbpm)</td>
<td>Provides integration with jBPM (Business Process Management).</td>
</tr>
<tr>
<td>jbp:connectionURL</td>
<td></td>
</tr>
<tr>
<td><strong>JCache</strong> (camel-jcache)</td>
<td>To perform caching operations using JSR107/ JCache as cache implementation.</td>
</tr>
<tr>
<td>jcache:cacheName</td>
<td></td>
</tr>
<tr>
<td><strong>JCR</strong> (camel-jcr)</td>
<td>To add/read nodes to/from a JCR compliant content repository.</td>
</tr>
<tr>
<td>jcr:host/base</td>
<td></td>
</tr>
<tr>
<td><strong>JDBC</strong> (camel-jdbc)</td>
<td>To access databases through JDBC where SQL queries are sent in the message body.</td>
</tr>
<tr>
<td>jdbc:dataSourceName</td>
<td></td>
</tr>
<tr>
<td>Component / ArtifactId / URI</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Jetty 9 (camel-jetty9)</td>
<td>Provides HTTP-based endpoints for consuming and producing HTTP requests.</td>
</tr>
<tr>
<td>jetty:httpUri</td>
<td></td>
</tr>
<tr>
<td>JGroups (camel-jgroups)</td>
<td>Provides exchange of messages between Camel and JGroups clusters.</td>
</tr>
<tr>
<td>jgroups:clusterName</td>
<td></td>
</tr>
<tr>
<td>JMS (camel-jms)</td>
<td>Allows messages to be sent to, or consumed from a JMS Queue or Topic.</td>
</tr>
<tr>
<td>jms:destinationType:destinatonName</td>
<td></td>
</tr>
<tr>
<td>JMX (camel-jmx)</td>
<td>To receive JMX notifications.</td>
</tr>
<tr>
<td>jmx:serverURL</td>
<td></td>
</tr>
<tr>
<td>JOLT (camel-jolt)</td>
<td>To process JSON messages using a JOLT specification (such as JSON-JSON transformation).</td>
</tr>
<tr>
<td>jolt:resourceUri</td>
<td></td>
</tr>
<tr>
<td>JPA (camel-jpa)</td>
<td>To store and retrieve Java objects from databases using JPA.</td>
</tr>
<tr>
<td>jpa:entityType</td>
<td></td>
</tr>
<tr>
<td>Jsch (camel-jsch)</td>
<td>To copy files using the secure copy protocol (SCP).</td>
</tr>
<tr>
<td>scp:host:port/directoryName</td>
<td></td>
</tr>
<tr>
<td>Kafka (camel-kafka)</td>
<td>Allows messages to be sent to, or consumed from Apache Kafka brokers.</td>
</tr>
<tr>
<td>kafka:topic</td>
<td></td>
</tr>
<tr>
<td>Krati (camel-krati)</td>
<td>Allows the use of krati datastores and datasets inside Camel.</td>
</tr>
<tr>
<td>krati:path</td>
<td></td>
</tr>
<tr>
<td>Kubernetes (camel-kubernetes)</td>
<td>To work with Kubernetes PaaS.</td>
</tr>
<tr>
<td>kubernetes:masterUrl</td>
<td></td>
</tr>
<tr>
<td>Lucene (camel-lucene)</td>
<td>To insert or query from Apache Lucene databases.</td>
</tr>
<tr>
<td>lucene:host:operation</td>
<td></td>
</tr>
<tr>
<td>Mail (camel-mail)</td>
<td>To send or receive emails using imap/pop3 or smtp protocols.</td>
</tr>
<tr>
<td>imap:host:port</td>
<td></td>
</tr>
<tr>
<td>Mina2 (camel-mina2)</td>
<td>Socket level networking using TCP or UDP with the Apache Mina 2.x library.</td>
</tr>
<tr>
<td>mina2:protocol:host:port</td>
<td></td>
</tr>
<tr>
<td>MLLLP (camel-mllp)</td>
<td>Provides functionality required by Healthcare providers to communicate with other systems using the MLLP protocol.</td>
</tr>
<tr>
<td>mllp:hostname:port</td>
<td></td>
</tr>
<tr>
<td>Component / ArtifactId / URI</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Mock (camel-core)mock:name</td>
<td>For testing routes and mediation rules using mocks.</td>
</tr>
<tr>
<td>MongoDB (camel-mongodb)mongodb:connectionBean</td>
<td>For working with documents stored in MongoDB database.</td>
</tr>
<tr>
<td>MongoDB GridFS (camel-mongodb-gridfs)mongodb-gridfs:connectionBean</td>
<td>For working with MongoDB GridFS.</td>
</tr>
<tr>
<td>MQTT (camel-mqtt)mqtt:name</td>
<td>For communicating with MQTT M2M message brokers using FuseSource MQTT Client.</td>
</tr>
<tr>
<td>Mustache (camel-mustache)mustache:resourceUri</td>
<td>Transforms the message using a Mustache template.</td>
</tr>
<tr>
<td>MyBatis (camel-mybatis)mybatis:statement</td>
<td>Performs a query, poll, insert, update or delete in a relational database using MyBatis.</td>
</tr>
<tr>
<td>Nats (camel-nats)nats:servers</td>
<td>Produces and consumes messages from NATS.</td>
</tr>
<tr>
<td>Netty (camel-netty)netty:protocol:host:port</td>
<td>Socket level networking using TCP or UDP with the Netty 3.x library. This component is deprecated.</td>
</tr>
<tr>
<td>OpenShift (camel-openshift)openshift:clientId</td>
<td>To Manage your Openshift 2.x applications. This component is deprecated.</td>
</tr>
<tr>
<td>OptaPlanner (camel-optaplanner)optaplanner:configFile</td>
<td>Solves the planning problem contained in a message with OptaPlanner.</td>
</tr>
<tr>
<td>Paho (camel-paho)paho:topic</td>
<td>For communicating with MQTT M2M message brokers using Eclipse Paho MQTT Client.</td>
</tr>
<tr>
<td>PDF (camel-pdf)pdf:operation</td>
<td>Provides the ability to create, modify or extract content from PDF documents.</td>
</tr>
<tr>
<td>Quartz (camel-quartz)quartz:groupName/timerName</td>
<td>Provides a scheduled delivery of messages using the Quartz 1.x scheduler. This component is deprecated.</td>
</tr>
<tr>
<td>Quartz2 (camel-quartz2)quartz2:groupName/triggerName</td>
<td>Provides a scheduled delivery of messages using the Quartz 2.x scheduler.</td>
</tr>
<tr>
<td>Component / ArtifactId / URI</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>RabbitMQ</strong> (camel-rabbitmq) rabbitmq:hostname:portNumber/exchangeName</td>
<td>To produce and consume messages from RabbitMQ instances.</td>
</tr>
<tr>
<td><strong>RMI</strong> (camel-rmi) rmi:hostname:port/name</td>
<td>For invoking Java RMI beans from Camel.</td>
</tr>
<tr>
<td><strong>RSS</strong> (camel-rss) rss:feedUri</td>
<td>For consuming RSS feeds.</td>
</tr>
<tr>
<td><strong>Salesforce</strong> (camel-salesforce)销售force:operationName:topicName</td>
<td>For integrating Camel with the massive Salesforce API.</td>
</tr>
<tr>
<td><strong>SAP NetWeaver</strong> (camel-sap-netweaver) sap-netweaver:url</td>
<td>To integrate with the SAP NetWeaver Gateway using HTTP transports.</td>
</tr>
<tr>
<td><strong>Servlet</strong> (camel-servlet) servlet:contextPath</td>
<td>To use a HTTP Servlet as entry for Camel routes when running in a servlet container.</td>
</tr>
<tr>
<td><strong>Simple JMS</strong> (camel-sjms) sjms:destinationType:destinationName</td>
<td>Allows messages to be sent to, or consumed from a JMS Queue or Topic (uses JMS 1.x API).</td>
</tr>
<tr>
<td><strong>Slack</strong> (camel-slack) slack:channel</td>
<td>To send messages to Slack.</td>
</tr>
<tr>
<td><strong>SMPP</strong> (camel-smpp) smpp:host:port</td>
<td>To send and receive SMS using a SMSC (Short Message Service Center).</td>
</tr>
<tr>
<td><strong>SNMP</strong> (camel-snmp) snmp:host:port</td>
<td>To poll SNMP capable devices or receiving traps.</td>
</tr>
<tr>
<td><strong>Solr</strong> (camel-solr) solr:url</td>
<td>To interface with an Apache Lucene Solr server.</td>
</tr>
<tr>
<td><strong>Splunk</strong> (camel-splunk) splunk:name</td>
<td>To publish or search for events in Splunk.</td>
</tr>
<tr>
<td><strong>Spring Batch</strong> (camel-spring-batch) spring-batch:jobName</td>
<td>To send messages to Spring Batch for further processing.</td>
</tr>
<tr>
<td><strong>Spring Integration</strong> (camel-spring-integration) spring-integration:defaultChannel</td>
<td>Bridges Camel with Spring Integration.</td>
</tr>
<tr>
<td>Component / ArtifactId / URI</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Spring LDAP</strong> (camel-spring-ldap)</td>
<td>To perform searches in LDAP servers using filters as the message payload.</td>
</tr>
<tr>
<td>spring-ldap:templateName</td>
<td></td>
</tr>
<tr>
<td><strong>Spring Redis</strong> (camel-spring-redis)</td>
<td>To send and receive messages from Redis.</td>
</tr>
<tr>
<td>spring-redis:host:port</td>
<td></td>
</tr>
<tr>
<td><strong>Spring WebService</strong> (camel-spring-ws)</td>
<td>Works for SOAP WebServices using Spring WebServices.</td>
</tr>
<tr>
<td>spring-ws:type:lookupKey:web ServiceEndpointUri</td>
<td></td>
</tr>
<tr>
<td><strong>SQL</strong> (camel-sql)</td>
<td>To work with databases using JDBC SQL queries.</td>
</tr>
<tr>
<td>sql:query</td>
<td></td>
</tr>
<tr>
<td><strong>SSH</strong> (camel-ssh)</td>
<td>The ssh component enables access to SSH servers so that you can send an SSH command and process the response.</td>
</tr>
<tr>
<td>ssh:host:port</td>
<td></td>
</tr>
<tr>
<td><strong>StAX</strong> (camel-stax)</td>
<td>Allows messages to be process through a SAX ContentHandler.</td>
</tr>
<tr>
<td>stax:contentHandlerClass</td>
<td></td>
</tr>
<tr>
<td><strong>Stomp</strong> (camel-stomp)</td>
<td>For communicating with Stomp compliant message brokers.</td>
</tr>
<tr>
<td>stomp:destination</td>
<td></td>
</tr>
<tr>
<td><strong>Twitter</strong> (camel-twitter)</td>
<td>Integrates with Twitter to send tweets or search for tweets and more.</td>
</tr>
<tr>
<td>twitter:kind</td>
<td></td>
</tr>
<tr>
<td><strong>Undertow</strong> (camel-undertow)</td>
<td>Provides HTTP-based endpoints for consuming and producing HTTP requests.</td>
</tr>
<tr>
<td>undertow:httpURI</td>
<td></td>
</tr>
<tr>
<td><strong>Velocity</strong> (camel-velocity)</td>
<td>Transforms the message using a Velocity template.</td>
</tr>
<tr>
<td>velocity:resourceUri</td>
<td></td>
</tr>
<tr>
<td><strong>Vertx</strong> (camel-vertx)</td>
<td>For sending and receive messages from a vertx event bus.</td>
</tr>
<tr>
<td>vertx:address</td>
<td></td>
</tr>
<tr>
<td><strong>Weather</strong> (camel-weather)</td>
<td>Polls the weather information from Open Weather Map.</td>
</tr>
<tr>
<td>weather:name</td>
<td></td>
</tr>
<tr>
<td><strong>Yammer</strong> (camel-yammer)</td>
<td>To interact with the Yammer enterprise social network.</td>
</tr>
<tr>
<td>yammer:function</td>
<td></td>
</tr>
<tr>
<td><strong>ZooKeeper</strong> (camel-zooookeeper)</td>
<td>Allows interaction with a ZooKeeper cluster.</td>
</tr>
<tr>
<td>zookeeper:serverUrls/path</td>
<td></td>
</tr>
</tbody>
</table>
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
```
To call the `mq.properties` file, 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` on page 60.

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`:

```java
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](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` on page 246 and `cWMQ` on page 350.

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

<table>
<thead>
<tr>
<th>URI</th>
<th>URI of the messages to send or receive. It can be of different format:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- File: &quot;file:/&quot;,</td>
</tr>
<tr>
<td></td>
<td>- Database: &quot;jdbc:/&quot;,</td>
</tr>
<tr>
<td></td>
<td>- Protocols: &quot;ftp:/&quot;, &quot;http:/&quot;</td>
</tr>
</tbody>
</table>
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](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 224. |
| --- | --- |

**Note:**

If you want to use a custom component, select this check box and click the three-dot button to upload a jar file with your own component.
All the transitive dependencies of this custom component should be included in the jar file.

### Usage

**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

**Procedure**

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

**Procedure**

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.
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.

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

### Viewing code and executing the Route

#### Procedure

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

```java
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");
            }
            getCamelContext().get().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.

![Diagram of components](image)

Dropping and linking components

**Procedure**

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

**Procedure**

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`.

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.
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.

6. Press Ctrl+S to save your Route.

**Viewing code and executing the Route**

**Procedure**

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

```java
protected void initUriMap() {
    uriMap = new java.util.HashMap<String, String>();
    uriMap.put("Sender", "http://indus@degas/remp/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");
    }
    getCamelContext().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

Procedure

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.

![Create Route Resource](image)

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

```xml
declare namespace ns0="http://com.sap/b";
<people>
{
for $p in /people//person
return
<person>
```
4. Press **Ctrl+S** to save your Route Resource.

**Dropping and linking the components**

**About this task**

![Diagram of components]

**Procedure**

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**

**Procedure**

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

![Basic settings for cFile]

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.

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.

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](http://camel.apache.org/xquery-endpoint.html).
9. Double-click the **cProcessor** component to open its **Basic settings** view in the **Component** tab.

![cProcessor Basic settings](image)

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

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

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

**Viewing code and executing the Route**

**Procedure**

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(urlMap.get("Message_source_cFile_1"))
                .routeId("Message_source_cFile_1")
                .to(urlMap
                    .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:
" + 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.

![Job cMessagingEndpoint](image)

**cMock**

Simulates message generation and message endpoints for testing Routes and mediation rules.

**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

<table>
<thead>
<tr>
<th><strong>Validate message count</strong></th>
<th>Select this check box to test that the correct number of messages are received on each endpoint. Specify the expected message number in the <strong>expected number</strong> field.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Header / Validate message header</strong></td>
<td>Select this check box to test that the correct message header is received on each endpoint.</td>
</tr>
<tr>
<td><strong>Header / Use File</strong></td>
<td>This option appears when the <strong>Validate message header</strong> 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 <strong>File name</strong> field.</td>
</tr>
<tr>
<td><strong>Header / Use Inline Table</strong></td>
<td>This option appears when the <strong>Validate message header</strong> 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.</td>
</tr>
<tr>
<td><strong>Body / Validate message bodies</strong></td>
<td>Select this check box to test that the correct message bodies are received on each endpoint.</td>
</tr>
<tr>
<td><strong>Body / Use File</strong></td>
<td>This option appears when the <strong>Validate message bodies</strong> 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 <strong>File name</strong> field.</td>
</tr>
<tr>
<td><strong>Body / Use Inline Table</strong></td>
<td>This option appears when the <strong>Validate message bodies</strong> 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.</td>
</tr>
<tr>
<td>Simulate</td>
<td>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.</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Wait time (in millis)</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Usage**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cMock can be a start, middle or end component in a Route.</th>
</tr>
</thead>
</table>

**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

Encapsulates a set of configuration parameters to connect to a MQ server. The connection can be called by multiple cJMS, cWMQ, cAMQP or cMQTT components in a Route.
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:

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

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

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

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

```properties
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:

```properties
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` on page 246.

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 247.  
For more information on connecting to WebSphere MQ, see the online documentation on Talend Help Center (https://help.talend.com). |
| **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.commontservices.jar`, `com.ibm.mq.headers.jar`, `com.ibm.mq.jms.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** on page 173.
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

<table>
<thead>
<tr>
<th>ConnectionFactory</th>
<th>This option appears when Use Connection Factory is selected. Click [...] and select a connection factory to be used for handling messages.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic Name</td>
<td>Type in a name for the message topic in the message broker.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>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 on page 246.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>Due to license incompatibility, one or more JARs required to use this component are not</td>
</tr>
</tbody>
</table>
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

About this task

Procedure

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

Procedure

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

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.

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.
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.

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.

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

**Viewing the code**

**Procedure**

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

About this task

Procedure

1. From the Palette, drag and drop a cMQConnectionFactory, a cMQTT, and a Log 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

Procedure

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

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

**Viewing the code and executing the Route**

**Procedure**

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

```java
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.

3. Execute the producer Route. The logs of the message exchange are printed in the console.
4. In the consumer Route, the messages are consumed and shown in the execution console.

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

Routes messages to a number of endpoints at one go and process them in different ways.

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**

<table>
<thead>
<tr>
<th>cMulticast</th>
<th>cMulticast can be used as a middle or end component in a Route.</th>
</tr>
</thead>
</table>

**Connections**

<table>
<thead>
<tr>
<th>Route</th>
<th>Route: Select this link to route the message(s) from the sender to the next endpoint.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndBlock</td>
<td>EndBlock: Select this link to route the message(s) from the end block of the Route to the next endpoint.</td>
</tr>
</tbody>
</table>

**Limitation**

| n/a | 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

Procedure

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

Procedure

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

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.

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 `Processor` component labeled `Monitor_source` to open its `Basic settings` view in the `Component` tab.

![Monitor_source component](image)

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 `SetBody` component labeled `Set_new_body` to open its `Basic settings` view in the `Component` tab.

![Set_new_body component](image)
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

Procedure

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

```java
public RouteBuilder route() {
    return new RouteBuilder()
        .process(new org.apache.camel.Exchange) {
            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.Exchange) {
        public void process(
            org.apache.camel.Exchange exchange)
        throws Exception {
            System.out.println("The endblock message is: ");
            exchange.getIn()
                .getBody(String.class));
    }
}
}
```

```
from(uriMap.get("direct_a_oMessagingEndpoint_1")
    .routeId("direct_a_oMessagingEndpoint_1")
    .simple("message A")
    .id("cSetBody_1")
    .process(new org.apache.camel.Exchange) {
        public void process(
            org.apache.camel.Exchange exchange)
        throws Exception {
            System.out.println("direct a just downloaded: ");
            exchange.getIn()
                .getBody(String.class));
    }
}
```

```
from(uriMap.get("direct_a_oMessagingEndpoint_2")
    .routeId("direct_a_oMessagingEndpoint_2")
    .simple("message B")
    .id("cSetBody_2")
    .process(new org.apache.camel.Exchange) {
        public void process(
            org.apache.camel.Exchange exchange)
        throws Exception {
            System.out.println("direct b just downloaded: ");
            exchange.getIn()
                .getBody(String.class));
    }
}
```

```
from(uriMap.get("direct_a_oMessagingEndpoint_3")
    .routeId("direct_a_oMessagingEndpoint_3")
    .simple("message C")
    .id("cSetBody_3")
    .process(new org.apache.camel.Exchange) {
        public void process(
            org.apache.camel.Exchange exchange)
        throws Exception {
            System.out.println("direct c just downloaded: ");
            exchange.getIn()
                .getBody(String.class));
    }
}
```

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.

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

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

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>Click the plus button to add as many lines as needed in the table to define the exceptions to be caught.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set a redelivering tries count</td>
<td>Select this check box to set the maximum redelivering tries in the Maximum redelivering tries field.</td>
</tr>
</tbody>
</table>
### 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

Procedure

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

Procedure

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

![Basic settings for cOnException component](image)

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.
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.

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:

```java
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.

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

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

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

**Viewing code and executing the Route**

**Procedure**

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

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**

Splits message routing into a series of independent processing stages.

**cPipesAndFilters** 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**

<table>
<thead>
<tr>
<th>URI list</th>
<th>Click the plus button to add new lines for URIs that identify endpoints.</th>
</tr>
</thead>
</table>

**Usage**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cPipesAndFilters is usually used in the middle of a Route.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Limitation</th>
<th>n/a</th>
</tr>
</thead>
</table>

**Scenario: Using cPipesAndFilters to process the task in sequence**

This scenario applies only to a Talend solution with ESB.

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

Procedure

1. From the Connectivity folder of the Palette, drop two cFile components onto the design workspace, one as the message sender and the other as the message receiver, and label them Sender and Receiver respectively to better identify their roles in the Route.
2. From the Routing folder, drop one cPipesAndFilters component onto the design workspace, between the two cFile components.
3. From the Core folder, drop two cMessagingEndpoint components onto the design workspace, one as the endpoint of stage A and the other as the endpoint of stage B, and label them Stage_A and Stage_B respectively to better identify their roles in the Route.
4. From the Custom folder, drop three cProcessor components onto the design workspace to monitor messages received on the receiver, stage A and stage B endpoints respectively, and label them Monitor_Receiver, Monitor_stage_A, and Monitor_stage_B respectively to better identify their roles in the Route.
5. Right-click the cFile component labeled Sender, select Row > Route from the contextual menu, and click the cPipesAndFilters component. Repeat this step to set up the rest Row > Route connections, as shown above.

Configuring the components

Procedure

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

   ![Sender(cFile_1)](image)

2. In the Path field, fill in or browse to the path to the folder that holds the source files.
3. From the Encoding list, select the encoding type of your source files. Leave the other parameters as they are.
4. Repeat these steps to define the path to the output files and the output encoding type in the Basic settings view of the cFile component labeled Receiver.
5. Double-click the cPipesAndFilters component to open its Basic settings view in the Component tab.
6. Click the plus button to add two lines to the URI list table, and fill the first line with "direct:a" and the second line with "direct:b" to define the URIs of stage A and stage B that the messages will undergo.

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

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

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

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

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

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

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

9. Press Ctrl+S to save your Route.

Viewing code and executing the Route

Procedure

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

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

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

2. Press F6 to run your Route.

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

cProcessor

Quickly whirls up some code. If the code in the inner class is complicated, it is recommended 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

<table>
<thead>
<tr>
<th>Imports</th>
<th>Enter the Java code that helps to import, if necessary, external libraries used in the Code box.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Type in the Java code you want to implement.</td>
</tr>
</tbody>
</table>

Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cProcessor is used as a middle or end component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Related scenario:

For a related scenario, see Scenario: Intercepting several routes and redirect them in a single new route on page 168 of cIntercept on page 167.

cRecipientList

Routes messages to a number of dynamically specified recipients.

cRecipientList 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

<table>
<thead>
<tr>
<th>Language</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Type in the expression that returns multiple endpoints.</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>Use Result Class Type</td>
<td>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.</td>
</tr>
<tr>
<td>Add Namespaces</td>
<td>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.</td>
</tr>
<tr>
<td>Use Delimiter</td>
<td>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.</td>
</tr>
<tr>
<td>Use Strategy</td>
<td>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.</td>
</tr>
<tr>
<td>Parallel Processing</td>
<td>Select this check box to send the message to the recipients simultaneously.</td>
</tr>
<tr>
<td>Use ExecutorService</td>
<td>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.</td>
</tr>
<tr>
<td>Stop On Exception</td>
<td>Select this check box to stop processing immediately when an exception occurred.</td>
</tr>
<tr>
<td>Ignore Invalid Endpoints</td>
<td>Select this check box to ignore invalid endpoints.</td>
</tr>
<tr>
<td><strong>Streaming</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Timeout</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Use On-Prepare Processor</strong></td>
<td>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 \texttt{cBeanRegister} to register a Java bean as a processor.</td>
</tr>
<tr>
<td><strong>Share Unit of Work</strong></td>
<td>Select this check box to share the unit of work between the parent exchange and each recipient exchange. See the same option of \texttt{cSplitter} for more information.</td>
</tr>
</tbody>
</table>

**Usage**

| **Usage rule** | \texttt{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 \texttt{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**

**Procedure**

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**

**Procedure**

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

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.

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.
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"));
```

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**

**Procedure**

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

Routes the message 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

<table>
<thead>
<tr>
<th>Header name</th>
<th>Type in name of the message header as defined in the preceding <code>cSetHeader</code> component, <code>mySlip</code> by default. The header should carry a list of endpoint URIs you wish each message to be routed to.</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI delimiter</td>
<td>Delimiter used to separate multiple endpoint URIs carried in the message header, comma (,) by default.</td>
</tr>
</tbody>
</table>

### 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

**About this task**

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.
Procedure

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

Procedure

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

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.
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`.

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:

```java
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.

```java
System.out.println("Message received on endpoint b: "+
exchange.getIn().getHeader("CamelFileName");
```
9. Press Ctrl+S to save your Route.

Viewing code and executing the Route

Procedure

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

```java
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.

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:_

```xml
<person>
  <firstName>Nicolas</firstName>
  <lastName>Yang</lastName>
  <city>Beijing</city>
</person>
```

_Paris.xml:_

```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_.

```java
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";
        }...
    }
}
```
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

About this task

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 as they are in the previous scenario.

Procedure

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

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

Procedure

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

```java
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")
                .recuringSlip(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 URLs 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.

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

Produces and consumes 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**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td>Type in any string that uniquely identifies the endpoint.</td>
</tr>
<tr>
<td><strong>Specify maximum capacity size</strong></td>
<td>Select this check box to set the maximum number of messages that the SEDA queue can hold. Specify the number in the <strong>Size</strong> field.</td>
</tr>
<tr>
<td><strong>Concurrent consumers</strong></td>
<td>Specify the number of concurrent threads processing exchanges.</td>
</tr>
<tr>
<td><strong>Wait for task to complete</strong></td>
<td>Specify whether the caller should wait for the asynchronous task to complete or not before continuing. Select from <strong>Always</strong>, <strong>Never</strong> or <strong>IfReplyExpected</strong>. The default option is <strong>IfReplyExpected</strong> which means the caller will only wait if the message is Request-Reply based. For more information about this option, see the site <a href="http://camel.apache.org/async.html">http://camel.apache.org/async.html</a>.</td>
</tr>
<tr>
<td><strong>Timeout</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Use multiple consumers</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Limit concurrent consumers</strong></td>
<td>Whether to limit the number of concurrent consumers to the maximum of 500. By default, an...</td>
</tr>
</tbody>
</table>
exception will be thrown if a SEDA endpoint is configured with a greater number.

<table>
<thead>
<tr>
<th>Block when full</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poll timeout</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

Procedure

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

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.
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.

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

   ```xml
   <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

About this task

![Diagram showing component placement and linking]

Procedure

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

Procedure

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

   ![Starter(cTimer_1) settings](image)

   - **Period**: 1000
   - **Repeat**: "{{repeat.count}}"
   - **Delay**: 1000
   - **Daemon**: checked
   - **Set Schedule Time**: unchecked

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.
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_Direct:**

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

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`.

Select the node `Direct_consumer` for `VM_producer`.

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

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`. 
Give the name "direct" to Direct_consumer.

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

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

Procedure

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

```java
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")
    from(uriMap.get("SEDA_consumer_cSEDA_1"))
        .routeId("Direct_consumer_cDirect_2")
        .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.

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

Sets 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

<table>
<thead>
<tr>
<th>Language</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Type in the expression to set the message content.</td>
</tr>
<tr>
<td>Correlation expression/Add Namespaces</td>
<td>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.</td>
</tr>
</tbody>
</table>

Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cSetBody is used as a middle component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

**About this task**

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.

**Procedure**

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

**Procedure**

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

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:

   ```java
   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

Procedure

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

   ```java
   public void initRoute() throws Exception {
       routeBuilder = new org.apache.camel.builder.RouteBuilder()
   }
   public void configure() throws Exception {
       from(urlMap.get("Sender")).routeId("Sender").setBody()
           .xpath("/people/person").id("cSetBody_1").to(
               urlMap.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.

RESULT: The XML files are sent to the receiver, where `person` has become the root element of each file.

**cSetHeader**

Sets headers or customizes the default headers, if any, on each message sent to it for subsequent message processing.

**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

<table>
<thead>
<tr>
<th>Headers</th>
<th>Click [+] to add as many headers as required to the table.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong>: Type in a name for the message header. The header name <code>CorrelationID</code> is reserved. The value of this header will be overridden by the correlation ID of the message if it exist.</td>
<td></td>
</tr>
<tr>
<td><strong>Language</strong>: Select the language of the expression you use from <code>None</code>, <code>Bean</code>, <code>Constant</code>, <code>CorrelationID</code>, <code>EL</code>, <code>Groovy</code>, <code>Header</code>, <code>JavaScript</code>, <code>JoSQL</code>, <code>JSonPath</code>, <code>JXPath</code>, <code>MVEL</code>, <code>OGNL</code>, <code>PHP</code>, <code>Property</code>, <code>Python</code>, <code>Ruby</code>, <code>Simple</code>, <code>SpEL</code>, <code>SQL</code>, <code>XPath</code>, and <code>XQuery</code>. Select <code>Bean</code> if you want to call a predefined Java Bean to return the header value. Select <code>CorrelationID</code> to use the existing correlation ID of the message as the header value if the correlation ID is available in the closest <code>cSOAP</code> connected to this component. For more information about the <code>cSOAP</code> component, see <code>cSOAP</code> on page 79.</td>
<td></td>
</tr>
<tr>
<td><strong>Value</strong>: 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 <code>beans.BEAN_NAME</code>. If <code>CorrelationID</code> is selected in the <code>Language</code> list, this field is grayed out. The existing correlation ID from the closest <code>cSOAP</code> connected to this component will be set as the header value. For more information about the <code>cSOAP</code> component, see <code>cSOAP</code> on page 79.</td>
<td></td>
</tr>
</tbody>
</table>
### 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

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cSetHeader is used as a middle component in a Route.</th>
</tr>
</thead>
</table>

### Limitation

<table>
<thead>
<tr>
<th>Limitation</th>
<th>n/a</th>
</tr>
</thead>
</table>

#### Scenario: Splitting a message and renaming the sub-messages according to contained information

This scenario applies only to a Talend solution with ESB.

In this scenario, a file message containing people information is split into sub-messages. Each sub-messages is renamed according the city name it contains, and then routed to another endpoint.

The following is the example XML file used in this use case:

```xml
<people>
  <person>
    <firstName>Pierre</firstName>
    <lastName>Dubois</lastName>
    <city>Paris</city>
  </person>
  <person>
    <firstName>Nicolas</firstName>
    <lastName>Yang</lastName>
    <city>Beijing</city>
  </person>
  <person>
    <firstName>Ellen</firstName>
    <lastName>Ripley</lastName>
    <city>Washington</city>
  </person>
</people>
```
A predefined Java Bean, `setFileNames`, is called by the `cSetHeader` component used in this use case to define a file name for each message according to the city name it contains. For more information about creating and using Java Beans, see *Talend Studio User Guide*.

```java
package beans;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class setFileNames {
    public String getCityName(Document document) {
        NodeList cities = document.getDocumentElement().getElementsByTagName("city");
        Element city = (Element) cities.item(0);
        String textContent = city.getTextContent();
        return textContent + ".xml";
    }
}
```

Dropping and linking the components

**About this task**

This use case uses two `cFile` components, one as the message sender and the other as the receiver, a `cSplitter` component to split the source message into sub-messages, a `cSetHeader` component to rename each sub-message, and a `cProcessor` component to display the file name of each message routed to the receiver.

**Procedure**

1. From the Palette, expand the Connectivity folder, and drop two `cFile` components onto the design workspace.
2. From the Routing folder, drop a `cSplitter` component onto the design workspace, between the two `cFile` components.
3. From the Core folder, drop a `cSetHeader` component onto the design workspace, between the `cSplitter` component and the receiving `cFile` component.
4. Right-click the first cFile component, select Row > Route from the contextual menu and click the cSplitter component.

5. Right-click the cSplitter component, select Row > Split from the contextual menu and click the cSetHeader component.

6. Right-click the cSetHeader component, select Row > Route from the contextual menu and click the second cFile component.

7. Right-click the second cFile component, select Row > Route from the contextual menu and click the cProcessor component.

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

Configuring the components and connections

Procedure

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

![Image of Sender(cFile_1) component settings]

2. In the Path field, fill in or browse to the path to the folder that holds the source files. From the Encoding list, select the encoding type of your source files. In the FileName field, type in the file name of the source message. You can skip this step if the source folder contains only one file.

3. Repeat steps 1 and 2 above to define the output file path and encoding type in the Basic settings view of the other cFile component, which is labeled Receiver. Leave the FileName field blank.
4. Double-click the cSplitter component to display its Basic settings view in the Component tab. In this use case, as we want to split the message into sub-messages at each person node of the XML file. Select XPath in the Language list and type in "/people/person") in the Expression field.

5. Double-click the cSetHeader component, which is labeled Set_file_name to display its Basic settings view in the Component tab.
6. Click [+] to add a row to the **Headers** table.
   
   In the **Name** field, type in the name of the header you want to give to the messages.
   
   Here, as we want to define the file name for each incoming message, fill in "CamelFileName" as the header name.
   
   Select **Bean** in the **Language** field and type in the name of the predefined Java Bean in the **Value** field, `beans.setFileNames.class` in this use case.

7. Double-click the **cProcessor** component to display its **Basic settings** view in the **Component** tab, and customize the code so that the console will display information the way you wish.
   
   In this use case, we want to display the file name each message routed to the receiving endpoint, so we customize the code as follows:
   
   ```
   System.out.println("File received: "+
   exchange.getIn().getHeader("CamelFileName"));
   ```

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

**Viewing code and executing the Route**

**Procedure**

1. Click the **Code** tab at the bottom of the design workspace to have a look at the generated code.
As shown in the code, a message route is built from one endpoint to another, and while in routing, the source message is split according to the condition xpath("/people/person") by cSplitter_1, and each sub-message is given a header named CamelFileName, the value of which is returned by method(beans.setFileNames.class).

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

RESULT: The source file message is split into sub-messages and each sub-message is renamed after the city name it contains and routed to the receiving endpoint.

Related scenarios

For more scenarios, see:

Scenario: Using cTalendJob to call a DI Job on page 325

Scenario 1: Routing a message consecutively to a series of endpoints on page 283

Scenario 2: Routing each message conditionally to a series of endpoints on page 287

cSetProperty

Sets properties for each message sent to it for subsequent message processing.

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

<table>
<thead>
<tr>
<th>Properties</th>
<th>Click [+] to add as many properties as required to the table:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Name</strong>: define the name of the property.</td>
</tr>
<tr>
<td></td>
<td><strong>Language</strong>: 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.</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong>: enter the value of the property.</td>
</tr>
<tr>
<td>Add Namespaces</td>
<td>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.</td>
</tr>
</tbody>
</table>

Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cSetProperty is used as a middle component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Related scenarios

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

cSplitter

Splits a message into several sub-messages so that they can be handled and treated differently in individual routes.
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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>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.</td>
</tr>
<tr>
<td>Correlation expression/Expression</td>
<td>Type in the expression to use to split the messages.</td>
</tr>
<tr>
<td>Correlation expression/Use Result Class Type</td>
<td>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&gt;NodeList will be used if not specified.</td>
</tr>
<tr>
<td>Correlation expression/Add Namespaces</td>
<td>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.</td>
</tr>
<tr>
<td>Use Strategy</td>
<td>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...</td>
</tr>
</tbody>
</table>
replies will be aggregated in the same order as they were split.

<table>
<thead>
<tr>
<th>Parameters/Parallel Processing</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters/Stop on Exception</td>
<td>Select this check box to stop processing immediately when an exception occurs.</td>
</tr>
<tr>
<td>Parameters/Streaming</td>
<td>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.</td>
</tr>
<tr>
<td>Parameters/Share Unit of Work</td>
<td>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 <a href="http://camel.apache.org/splitter.html">http://camel.apache.org/splitter.html</a>.</td>
</tr>
<tr>
<td>Parameters/Timeout</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Usage**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th><strong>cSplitter</strong> is used as a middle component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connections</td>
<td><strong>split</strong>: Select this link to route the split messages to the next endpoint.</td>
</tr>
<tr>
<td></td>
<td><strong>Route</strong>: Select this link to route all the messages from the sender to the next endpoint.</td>
</tr>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

| Limitation | n/a |
Scenario: Using cSplitter to split a message and aggregate replies from sub-messages

This scenario applies only to a Talend solution with ESB.

In this scenario, we will use the **cSplitter** component to split a message and aggregate the replies from sub-messages.

A predefined Java Bean, *AppendAggregator* will be called to as the strategy to aggregate the replies from sub-messages. For more information about creating and using Java beans, see *Talend Studio User Guide*.

```java
package beans;
import org.apache.camel.Exchange;
import org.apache.camel.processor.aggregate.AggregationStrategy;

public class AppendAggregator implements AggregationStrategy {

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

### Dropping and linking the components

**About this task**
**Procedure**

1. From the **Palette**, expand the **Custom** folder. Drag and drop a **cBeanRegister** component onto the design workspace.
2. From the **Orchestration** folder, drag and drop a **cTimer** components onto the design workspace.
3. From the **Core** folder, drag and drop two **cSetBody** and two **cDirect** components onto the design workspace.
4. From the **Routing** folder, drag and drop a **cSplitter** component onto the design workspace.
5. From the **Miscellaneous** folder, drag and drop two **cLog** components onto the design workspace.
6. Label the components to better identify their roles in the Route as shown above.
7. Right-click the **cSplitter** component, select **Row > Split** from the contextual menu and click the **cDirect** component labeled **directA_sender**.
8. Right-click the **cSplitter** component again, select **Row > Route** from the contextual menu and click the **cLog** component labeled **Log_Finished**.
9. In the same way, link the other components in the Route using the **Row > Route** connection as shown above.

**Configuring the components and connections**

**Procedure**

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

![appendAggregator](attachment:image)

2. In the **Id** field, enter "**appendAggregator**". Select the **Simple** option and in the **Class Name** field, enter the name of the predefined Java bean, **beans.AppendAggregator** in this scenario, which will be called later by the **cSplitter** component.
3. Double-click the **cTimer** component labeled **Starter** to display its **Basic settings** view in the **Component** tab.
4. In the Repeat field, enter 1 to trigger the message exchange. Keep the default settings of the other options.

5. Double-click the cSetbody component labeled Set_body to display its Basic settings view in the Component tab.

6. Select SIMPLE in the Language list and enter "thing1, thing2, thing3" in the Expression field as the message body.

7. Configure the cSetbody component labeled Set_new_body in the same way to set new body to the sub-messages. In the Expression field, enter *** SPLIT: ${body} to add *** SPLIT: $ before the old message body.

8. Double-click the cSplitter component to display its Basic settings view in the Component tab.
9. In this use case, we want to split the message body into sub-messages using , as the separator. Select Simple in the Language list and enter $\{body\}$ in the Expression field. Select the Use Strategy check box. In the Strategy field, enter the Id of the preregistered Java bean appendAggregator, which will be used to aggregate the replies from sub-messages.

10. Double-click the cDirect component labeled directA_sender to display its Basic settings view in the Component tab.

11. Click [...] and select the directA_receiver node to receive the sub-messages.

12. Double-click the cDirect component labeled directA_receiver to display its Basic settings view in the Component tab.

13. In the Name field, type in "directA" to identify this endpoint.

14. Keep the default settings of the cLog components labeled Log_Finished and Log_Split. Log_Finished will log the aggregated replies from sub-messages. Log_Split will log the sub-messages.

15. Press Ctrl+S to save your Route.
Viewing code and executing the Route

Procedure

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

```java
public void configure() throws java.lang.Exception {
    from(uriMap.get("directA_receiver_cDirect_1")).
        routeId("directA_receiver_cDirect_1").setBody()
            .simple("*** SPLIT: ${body}").id("cSetBody_2")
            .to(uriMap.get("Log_Split_cLog_1"))
            .id("cLog_1");

    from(uriMap.get("Starter_cTimer_1")).routeId("Starter_cTimer_1")
        .setBody().simple("thing1, thing2, thing3").id("cSetBody_1")
        .split().simple("${body}")
        .aggregationStrategyRef("appendAggregator").id("cSplitter_1")
        .to(uriMap.get("directA_receiver_cDirect_1")).id("cDirect_2")
        .end()

        .to(uriMap.get("Log_Finished_cLog_2"))
        .id("cLog_2");
}
```

As shown in the code, the message route is built from one endpoint to another. The message sent to `cSplitter_1` is split by the method `split().simple("${body}")` and the replies from the sub-messages are aggregated by the method `aggregationStrategyRef("appendAggregator")`.

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 3654
[statistics] connected
[INFO ]: Log_Split_cLog_1 - Exchange[ExchangePattern: InOnly,
    BodyType: String, Body: *** SPLIT: thing1]
[INFO ]: Log_Split_cLog_1 - Exchange[ExchangePattern: InOnly,
    BodyType: String, Body: *** SPLIT: thing2]
[INFO ]: Log_Split_cLog_1 - Exchange[ExchangePattern: InOnly,
    BodyType: String, Body: *** SPLIT: thing3]
[INFO ]: Log_Finished_cLog_2 - Exchange[ExchangePattern: InOnly,
    BodyType: String, Body: *** SPLIT: thing1
    *** SPLIT: thing2
    *** SPLIT: thing3]
```

RESULT: The source message is split into sub-messages using `,` as the separator and each sub-message is given a new message body. The replies from the sub-messages are aggregated then into a single outgoing message from the `cSplitter` component.
cStop

Stops a message routing 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 on page 168 of cIntercept on page 167.

cTalendJob

Exchanges 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.

<table>
<thead>
<tr>
<th>Repository/Use Job Context</th>
<th>This field appears when <strong>Repository</strong> is selected. Select this check box to use the selected context on the Job side when executing the Job.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Repository/Job</strong></td>
<td>This field appears when <strong>Repository</strong> is selected. Click [...] to show the <strong>[Assign Job]</strong> wizard. Choose between <strong>Create a new Job and Assign it to this cTalendJob component</strong> and <strong>Assign an existing Job to this cTalendJob component</strong> and follow the prompts.</td>
</tr>
<tr>
<td><strong>Warning:</strong></td>
<td>When assigning an existing Job to cTalendJob, only the Jobs with the <strong>tRouteInput</strong> component can be selected. You can double click cTalendJob to open the referenced Job, or right-click cTalendJob and select <strong>Open Job in Integration</strong> in the context menu open it.</td>
</tr>
<tr>
<td><strong>Repository/Version</strong></td>
<td>This field appears when <strong>Repository</strong> is selected. Select the version of the Job if more than one version of the Job is available.</td>
</tr>
<tr>
<td><strong>Repository/Context</strong></td>
<td>This field appears when <strong>Repository</strong> is selected. Select from the list the context to use to execute the referenced Job. <strong>Warning:</strong></td>
</tr>
</tbody>
</table>
This option works only when **Use Selected Context** is selected.

<table>
<thead>
<tr>
<th><strong>External Jar/Library</strong></th>
<th>This field appears when <strong>External</strong> 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.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>External Jar/Job</strong></th>
<th>This field appears when <strong>External</strong> is selected. Type in the name of the package and the name of your Job separated by a point. For example: <code>route_project.txmmap_0_1.tXMLMap</code>. To get this naming, you can open the jar library of your Job, go to OSGI-INF &gt; blueprint and edit the job.xml file, the naming is available in a bean node like <code>&lt;bean id=&quot;job&quot; class=&quot;route_project.txmmap_0_1.tXMLMap&quot;/&gt;</code>.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>External Jar/Context</strong></th>
<th>This field appears when <strong>External</strong> is selected. Type in the name of the context to use to execute the referenced Job.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Context Param</strong></th>
<th>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.</th>
</tr>
</thead>
</table>

**Advanced settings**

<table>
<thead>
<tr>
<th><strong>Propagate Header</strong></th>
<th>Select this check box to pass the message header to the referenced Job as a context variable.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Fast Job Invocation</strong></th>
<th>Select this check box to bind the life cycle of the embedded Talend Job to the start and stop</th>
</tr>
</thead>
</table>
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 |

Scenario: Using cTalendJob to call a DI Job

This scenario applies only to a Talend solution with ESB.

In this scenario, we will build a Data Integration Job with context first. Then, we will create a Route with a cTalendJob component to call the Job and set the context variables in the Job.

Creating an Data Integration Job

In this section, we will build a Job named RouteCommunication to accept the message from a Route. For more information on how to build a Job, see Talend Studio User Guide.

Dropping and linking the components

About this task

Procedure

1. Drag and drop a tRouteInput and a tLogRow from the Palette onto the design workspace.
2. Right-click the tRouteInput component, select Row > Main from the contextual menu and click the tLogRow component.
Configuring the components

Procedure

1. Create two variables, header and body in the Default context group. Give the value world to the body variable. The header variable will receive its value propagated from the Route. For more information about context setup, see *Talend Studio User Guide*.

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

3. Click [...] next to Edit Schema. In the schema dialog box, click [+] to add two lines of *String* type and name them header and body respectively. Click OK to close the dialog box.
4. In the **Simple Expression** field for the **header** element, enter `context.header` to use the variable `header` in the context group which will be propagated from the Route.

   In the same field for the **body** element, enter `context.body` to use the variable `body` in the context group as the message body.

5. The **tLogRow** component will monitor the message exchanges and does not need any configuration.

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

Creating a Mediation Route

In this section, we will create a Route to send a message to the Job.

**Arranging the flow of the message**

**About this task**

![Diagram of message flow]

**Procedure**

1. Drag and drop a **cTimer**, a **cSetHeader**, and a **cTalendJob** from the **Palette** onto the design workspace.

2. Link the components with the **Row > Route** connection as shown above.

3. Label the components for better identification of their roles.

**Configuring how the message is processed**

**Procedure**

1. Double-click the **cTimer** component to open its **Basic settings** view in the **Component** tab.
2. Set the values for the **Period** (200), **Repeat** (1) and **Delay** (1000) fields as shown above to trigger a message exchange after a delay of 1000 milliseconds.

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

4. Click [+ ] to add a row to the **Headers** table.
   
   In the **Name** field, type in "header" as the header name.
   
   Select **Constant** in the **Language** list, and enter "FileName" in the **Value** field.

5. Double-click the **cTalendJob** component to display its **Basic settings** view in the **Component** tab.
6. Select **Repository** to call a Job from the repository.
7. In the **Repository Job** area, select **Use Selected Context**.
   Click [...] next to the **Job** field to open the [Assign Job] wizard. Select **Assign an existing Job to this cTalendJob component** and click **Next**.

   ![Assign Job](Call_DI_job_cTalendJob_1.png)

   In the **Job selection** view, select **RouteCommunication** that we just created in the **Job designs** tree view and click **Finish**.
RouteCommunication is now displayed in the Job field. By default, the latest version and the default context of it is selected.

8. Click [+ ] under the Context Param to add a row to it.

The Parameters list contains the variables in the default context group of the referenced Job. Select body in the list.

Enter "Hello World!" in the Values field. It will replace the value world that is defined in the Job context.

9. Click the Advanced settings view. Select the Propagate Header check box to pass the header that is defined in cSetHeader to the Job as a context variable.

10. Press Ctrl+S to save your Route.

Viewing the code and executing the Route

Procedure

1. Click the Code tab at the bottom of the design workspace to check the generated code.
As shown above, a message route is built from the Starter, set a message header by the cSetHeader, and then sent to cTalendJob for the execution of the Job.

2. Press F6 to execute the Route.

As shown above, the header and body of the message is printed in the execution console by the tLogRow. The header value is FileName. It is defined by the cSetHeader and passed to the Job as a context variable. The body value is Hello World!. It is defined by tRouteInput using the context variable body of the Job. The value for this variable is set as world in the Job, but changed to Hello World! in cTalendJob.

cThrottler

Limits the number of messages flowing to a specific endpoint in order to prevent it from getting overloaded.

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

<table>
<thead>
<tr>
<th>Request per period</th>
<th>The number of messages allowed to pass cThrottler within the defined time period.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set time period</td>
<td>Select this check box to set the value of the time period (in milliseconds) and enable throttling.</td>
</tr>
<tr>
<td>Use asynchronous delaying</td>
<td>If this check box is selected, any messages that are delayed will be routed asynchronously using a scheduled thread pool.</td>
</tr>
</tbody>
</table>

Usage

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connections</td>
<td>throttler: Select this link to route the throttled messages to the next endpoint.</td>
</tr>
<tr>
<td>Route</td>
<td>Select this link to route all the messages from the sender to the next endpoint.</td>
</tr>
<tr>
<td>Limitation</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Scenario: Throttling the message flow

This scenario applies only to a Talend solution with ESB.

In this scenario, a cThrottler component is used to reduce the number of messages flowing out within a time period.
To build the Route, do the following.

**Dropping and linking the components**

**Procedure**

1. Drag and drop the components from the Palette onto the workspace: cThrottler, cFile and two cProcessor. Change the label of the cFile component to Read_Output. Change the labels of the two cProcessor components to Print_File_Name and Print_File_Content.
2. Link Read_Output to cThrottler using a Row > Route connection.
3. Link cThrottler to Print_File_Name using a Row > Throttler connection, and to Print_File_Content using a Row > Route connection.

**Configuring the components**

**Procedure**

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

2. In the Path field, type in the path to the source message, for example, "E:/data/output". Keep the default settings for other fields.
3. Double-click cThrottler to open its Basic settings view in the Component tab.

4. In the Request per period field, type in the number of messages allowed to pass the throttler per period, for example, 1.
   In the Set time period field, type in the value of the period, for example, 8000.
5. Double-click Print_File_Name to open its Basic settings view in the Component tab.
6. In the **Code** box, enter the code below to get the name of the message that passes the throttler.

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

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

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

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

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

**Viewing the code and executing the Route**

**Procedure**

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

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

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

Schedules message exchanges in a Route.

**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**

<table>
<thead>
<tr>
<th><strong>Period</strong></th>
<th>Fill this field with an integer (in milliseconds) to generate message exchanges every period.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Repeat</strong></td>
<td>Specifies a maximum limit of message exchange numbers. A value of zero or negative will generate message exchanges forever.</td>
</tr>
<tr>
<td><strong>Delay</strong></td>
<td>The number of milliseconds to wait before the first message exchange is generated. This option should not be used with the <strong>Set Schedule Time</strong> option.</td>
</tr>
<tr>
<td><strong>Fixed Rate</strong></td>
<td>Select this check box to generate message exchanges at regular intervals, separated by the specified period.</td>
</tr>
<tr>
<td><strong>Daemon</strong></td>
<td>Specify whether the thread associated with the timer endpoint runs as a daemon.</td>
</tr>
<tr>
<td><strong>Set Schedule Time</strong></td>
<td>Select this check box to specify the time that the first message exchange should be generated. In the <strong>Time</strong> field, enter the time using the pattern <code>yyyyMMdd HH:mm:ss</code> or <code>yyyyMMdd'T'HH:mm:ss</code>.</td>
</tr>
</tbody>
</table>

**Usage**

<table>
<thead>
<tr>
<th><strong>Usage rule</strong></th>
<th><strong>cTimer</strong> can only be used as a start component in a Route.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Limitation</strong></td>
<td>n/a</td>
</tr>
</tbody>
</table>


Related Scenario:
For a related scenario, see Scenario: Using cDataset to create messages on page 111.

**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**

<table>
<thead>
<tr>
<th>Usage rule</th>
<th>cTry is used as a middle component in a Route.</th>
</tr>
</thead>
</table>
| **Connections** | Try: Select this link to isolate the part of your Route that is likely to throw an exception or exceptions.  
 **Note:**  
When the Try link is followed by multiple components, a compile error may occur showing "The method doCatch() is undefined for the type ExpressionNode". In this case, use a cJavaDSLProcessor component to end the Try block with the code .endDoTry() as a workaround. |
| | Catch: Select this link to catch any exception thrown in the Route.  
In the Exceptions field, type in an expression to filter the type of exception to catch.  
**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
--- | ---

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

This scenario applies only to a Talend solution with ESB.

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

![Diagram](image_url)

Dropping and linking components

Procedure

1. From the Connectivity folder of the Palette, drop two cFile components onto the design workspace, one as the message sender and the other as the message receiver.
2. From the Exception Handling folder, drop a cTry component onto the design workspace to build Try, Catch and Finally blocks.
3. From the Custom folder, drop two cProcessor components onto the design workspace.
4. Link the cFile component serving as message sender to the cTry component using a Row > Route connection.
5. Link the **cTry** component to one **cProcessor** using a **Row > Try** connection. This **cProcessor** component will throw an exception if any file coming via this connection does not contain the required content.

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

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

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

### Configuring the components and connections

#### Procedure

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

   ![Sender(cFile 1)](image)

   - In the **Path** field, fill in or browse to the path to the folder that holds the source files.
   - From the **Encoding** list, select the encoding type of your source files. Leave the other parameters as they are.
   - Repeat these step to define the output file path and encoding type in the **Basic settings** view of the other **cFile** component, which is labeled **Receiver**.

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

   ```java
   String body = exchange.getIn().getBody(String.class);
   System.out.println("Trying: "+body);
   Exception e = new Exception("Only 'Talend Integration Solutions' is acceptable. Please check the file:");
   if(!"Talend Integration Solutions".equals(body)){
     throw e;
   }else{
     System.out.println("File is good.");
   }
   ```
6. Click the catch connection and then the Component tab to open its Basic settings view, and fill the Expression field with an expression to specify the type of exception to catch. In this scenario, fill in Exception.class to catch any exception thrown.

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

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

8. Click Ctrl+S to save your Route.

Viewing code and executing the Route

Procedure

1. Click the Code tab at the bottom of the design workspace to check the generated code.
As shown above, while messages are routed from the sender to the receiver, \texttt{doTry()}, \texttt{doCatch()} and \texttt{doFinally()} blocks are built by \texttt{cTry_1}. Thus, when any file does not meet the content requirement, an exception is thrown and caught, before each file is finally routed to the receiver.

2. Press \texttt{F6} to execute the Route.

![Execution](image)

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

Produces and consumes 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**

<table>
<thead>
<tr>
<th>When using as a start component in a Route:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td>Type in any string that uniquely identifies the endpoint.</td>
</tr>
<tr>
<td><strong>Specify maximum capacity size</strong></td>
<td>Select this check box to set the maximum number of messages that the SEDA queue can hold. Specify the number in the <strong>Size</strong> field.</td>
</tr>
<tr>
<td><strong>Concurrent consumers</strong></td>
<td>Specify the number of concurrent threads processing exchanges.</td>
</tr>
<tr>
<td><strong>Wait for task to complete</strong></td>
<td>Specify whether the caller should wait for the asynchronous task to complete or not before continuing. Select from <strong>Always</strong>, <strong>Never</strong> or <strong>IfReplyExpected</strong>. The default option is <strong>IfReplyExpected</strong> which means the caller will only wait if the message is Request-Reply based. For more information about this option, see the site <a href="http://camel.apache.org/async.html">http://camel.apache.org/async.html</a>.</td>
</tr>
<tr>
<td><strong>Timeout</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Use multiple consumers</strong></td>
<td>Specifies whether multiple consumers are allowed. If enabled, you can use <em>cVM</em> for Publish-Subscribe messaging, which means you can</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Limit concurrent consumers</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block when full</td>
<td>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.</td>
</tr>
<tr>
<td>Poll timeout</td>
<td>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.</td>
</tr>
</tbody>
</table>

When using as a middle or end component in a Route:

<table>
<thead>
<tr>
<th>Input endpoint name</th>
<th>Select this check box to enter the name of the corresponding consumer in the Name field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Exist cVM</td>
<td>This option appears when the Input endpoint name check box is cleared. Click [...] and select the corresponding consumer in the dialog box.</td>
</tr>
<tr>
<td>Name</td>
<td>This option appears when the Input endpoint name check box is selected. Enter the name of the consumer in this field directly.</td>
</tr>
</tbody>
</table>
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 293.

cWireTap
Routes 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

<p>| 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. |</p>
<table>
<thead>
<tr>
<th><strong>Populate Type</strong></th>
<th>This option appears when the <strong>Populate new exchange</strong> check box is selected. The <strong>Populate Type</strong> is either <strong>Expression</strong> or <strong>Processor</strong>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expression</strong></td>
<td>Using expression allows you to set the message body of the new exchange.</td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td>Select the language of the expression you want to use to set the message body from <strong>None</strong>, <strong>Bean</strong>, <strong>Constant</strong>, <strong>CorrelationID</strong>, <strong>EL</strong>, <strong>Groovy</strong>, <strong>Header</strong>, <strong>JavaScript</strong>, <strong>JoSQL</strong>, <strong>JSONPath</strong>, <strong>JXPath</strong>, <strong>MVEL</strong>, <strong>OGNL</strong>, <strong>PHP</strong>, <strong>Property</strong>, <strong>Python</strong>, <strong>Ruby</strong>, <strong>Simple</strong>, <strong>SpEL</strong>, <strong>SQL</strong>, <strong>XPath</strong>, and <strong>XQuery</strong>.</td>
</tr>
<tr>
<td><strong>Expression TXT</strong></td>
<td>Enter the expression to set the message body.</td>
</tr>
<tr>
<td><strong>Processor</strong></td>
<td>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 <strong>Code</strong> field.</td>
</tr>
<tr>
<td><strong>Copy the original message</strong></td>
<td>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 <strong>InOnly</strong> for both conditions.</td>
</tr>
<tr>
<td><strong>Usage</strong></td>
<td><strong>Usage rule</strong></td>
</tr>
<tr>
<td><strong>cWireTap</strong></td>
<td>can be a middle component in a Route.</td>
</tr>
</tbody>
</table>
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

**Procedure**

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** component 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

**Procedure**

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

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:

   ```java
   System.out.println("\nMessage wiretapped: "+
extchange.getIn().getHeader("CamelFileName"));
   System.out.println("Message content: "+
extchange.getIn().getBody(String.class)+"\n");
   ```

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

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*.

   ```java
   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

Procedure

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

```java
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.

![Execution Result]

```
Message wiretapped: Hello.txt
Message content: Hello world!
Message on endpoint b: Hello.txt
Message on endpoint a: Hello.txt
```
cWMQ

Exchanges 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

<table>
<thead>
<tr>
<th>URI/Type</th>
<th>Select the messaging type, either queue or topic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI/Destination</td>
<td>Type in a name for the message queue or topic in the message broker.</td>
</tr>
<tr>
<td>ConnectionFactory</td>
<td>Select a WMQ connection factory to be used for handling messages from the drop-down list.</td>
</tr>
</tbody>
</table>

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 on page 246. |

<table>
<thead>
<tr>
<th>Limitation</th>
</tr>
</thead>
</table>

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](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** on page 246.

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 = 
```
Related scenario

For a related scenario, see Scenario 1: Sending and receiving a message from a JMS queue on page 173.