**Polling an empty directory (and send an empty message with null body) :**

|  |  |
| --- | --- |
| 1 | from('[file://temp?sendEmptyMessageWhenIdle=true](file:///\\temp?sendEmptyMessageWhenIdle=true)') |

**Stop a route :**

|  |  |
| --- | --- |
| 1 | .process(new Processor() { |
| 2 | public void process(Exchange exchange) throws Exception { | |

|  |  |  |
| --- | --- | --- |
| 3 | getContext().stopRoute('ROUTE\_ID'); | |
| 4 | } |

|  |  |
| --- | --- |
| 5 | }) |

**Access a property of the object in the body :**

admitting the object has a method named ‘getMydata()’ :

|  |  |
| --- | --- |
| 1 | new ValueBuilder(simple('${body.mydata}')).isEqualTo(...) |

**Define an aggregator :**

|  |  |  |
| --- | --- | --- |
| 1 | .aggregate(simple('${header.id}.substring(0,15)'), | |
| 2 | genericAggregationStrategy) |

|  |  |  |
| --- | --- | --- |
| 3 | .completionPredicate(header(Exchange.BATCH\_COMPLETE) | |
| 4 | .isEqualTo(Boolean.TRUE)) |

* '${header.id}.substring(0,15)' : flag to differenciate messages (here, the returned string is common to all messages, we aggregate them all)
* Exchange.BATCH\_COMPLETE : predicate indicating the end of polling (all files parsed for example)
* genericAggregationStrategy : above, an example of an aggregator grouping all messages’ contents in a list :

|  |  |  |
| --- | --- | --- |
| 01 | public class GenericAggregationStrategy implements AggregationStrategy { | |
| 02 | @SuppressWarnings('unchecked') |

|  |  |  |
| --- | --- | --- |
| 03 | public Exchange aggregate(Exchange oldExchange, Exchange newExchange) { | |
| 04 | if (oldExchange == null) { |

|  |  |  |
| --- | --- | --- |
| 05 | ArrayList<Object> list = new ArrayList<Object>(); | |
| 06 | list.add(newExchange.getIn().getBody()); |

|  |  |  |
| --- | --- | --- |
| 07 | newExchange.getIn().setBody(list); | |
| 08 | return newExchange; |

|  |  |
| --- | --- |
| 09 | } else { |
| 10 | Object oldIn = oldExchange.getIn().getBody(); | |

|  |  |
| --- | --- |
| 11 | ArrayList<Object> list = null; |
| 12 | if(oldIn instanceof ArrayList) { | |

|  |  |  |
| --- | --- | --- |
| 13 | list = (ArrayList<Object>) oldIn; | |
| 14 | } else { |

|  |  |  |
| --- | --- | --- |
| 15 | list = new ArrayList<Object>(); | |
| 16 | list.add(oldIn); |

|  |  |
| --- | --- |
| 17 | } |
| 18 | list.add(newExchange.getIn().getBody()); | |

|  |  |  |
| --- | --- | --- |
| 19 | newExchange.getIn().setBody(list); | |
| 20 | return newExchange; |

|  |  |  |
| --- | --- | --- |
| 21 | } | |
| 22 | } |

|  |  |
| --- | --- |
| 23 | } |

**Manually trigger an aggregation’s completion (whatever it is) :**

Send a message with the header Exchange.AGGREGATION\_COMPLETE\_ALL\_GROUPS = true  
It is possible to do from('bean:...'), knowing that the bean will be polled permanently (like with ‘file’) and re-instanciated each time.**Modify the message’s body** on a route, using :

|  |  |
| --- | --- |
| 1 | .transform(myExpression) |

with myExpression :

|  |  |
| --- | --- |
| 1 | public class MyExpression implements Expression { |
| 2 | public <T> T evaluate(Exchange exchange, Class<T> type) { | |

|  |  |
| --- | --- |
| 3 | MyBean newData = ...; |
| 4 | return exchange.getContext().getTypeConverter() | |

|  |  |  |
| --- | --- | --- |
| 5 | .convertTo(type, newData); | |
| 6 | } |

|  |  |
| --- | --- |
| 7 | } |

**Using JaxB :**

* on a route :

|  |  |
| --- | --- |
| 1 | .[un]marshal().jaxb('my.business\_classes.package') |

* with a configurable DataFormat :

|  |  |
| --- | --- |
| 1 | .[un]marshal(jaxbDataFormat) |

* with :

|  |  |  |
| --- | --- | --- |
| 1 | // indicate to Jaxb to not write XML prolog : | |
| 2 | JaxbDataFormat jaxbDataFormat = |

|  |  |  |
| --- | --- | --- |
| 3 | new JaxbDataFormat('my.business\_classes.package'); | |
| 4 | jaxb.setFragment(true); |

**General concepts for threads management :**

* a from(...) = a thread
* except for from('direct:...') wich creates a ‘named route’ with a unique identifier only callable by another route (in the same thread than the caller).
* The component .resequence().batch() creates a new thread to rethrow the messages.

**Define a shutdown strategy :**

|  |  |
| --- | --- |
| 1 | getContext().setShutdownStrategy(new MyShutdownStrategy(getContext())); |

With :

|  |  |  |
| --- | --- | --- |
| 01 | public class MyShutdownStrategy extends DefaultShutdownStrategy { | |
| 02 | protected CamelContext camelContext; |

|  |  |
| --- | --- |
| 03 | private long timeout = 1; |
| 04 | private TimeUnit timeUnit = TimeUnit.SECONDS; | |

|  |  |  |
| --- | --- | --- |
| 05 | public SpiralShutdownStrategy(CamelContext camelContext) { | |
| 06 | this.camelContext = camelContext; |

|  |  |  |
| --- | --- | --- |
| 07 | } | |
| 08 |  |

|  |  |
| --- | --- |
| 09 | @Override |
| 10 | public long getTimeout() { | |

|  |  |  |
| --- | --- | --- |
| 11 | return this.timeout; | |
| 12 | } |

|  |  |
| --- | --- |
| 13 |  |
| 14 | @Override | |

|  |  |  |
| --- | --- | --- |
| 15 | public TimeUnit getTimeUnit() { | |
| 16 | return this.timeUnit; |

|  |  |  |
| --- | --- | --- |
| 17 | } | |
| 18 |  |

|  |  |
| --- | --- |
| 19 | @Override |
| 20 | public CamelContext getCamelContext() { | |

|  |  |  |
| --- | --- | --- |
| 21 | return this.camelContext; | |
| 22 | } |

|  |  |
| --- | --- |
| 23 |  |
| 24 | /\*\* | |

|  |  |  |
| --- | --- | --- |
| 25 | \* To ensure shutdown | |
| 26 | \* |

|  |  |
| --- | --- |
| 27 | \*/ |
| 28 | @Override | |

|  |  |
| --- | --- |
| 29 | public void suspend(CamelContext context, |
| 30 | List<RouteStartupOrder> routes) throws Exception { | |

|  |  |  |
| --- | --- | --- |
| 31 | doShutdown(context, routes, getTimeout(), | |
| 32 | getTimeUnit(), false, false, false); |

|  |  |  |
| --- | --- | --- |
| 33 | } | |
| 34 |  |

|  |  |
| --- | --- |
| 35 | /\*\* |
| 36 | \* To ensure shutdown | |

|  |  |
| --- | --- |
| 37 | \* |
| 38 | \*/ | |

|  |  |
| --- | --- |
| 39 | @Override |
| 40 | public void shutdown(CamelContext context, | |

|  |  |  |
| --- | --- | --- |
| 41 | List<RouteStartupOrder> routes, long timeout, | |
| 42 | TimeUnit timeUnit) throws Exception { |

|  |  |
| --- | --- |
| 43 | doShutdown(context, routes, this.timeout, |
| 44 | this.timeUnit, false, false, false); |

|  |  |  |
| --- | --- | --- |
| 45 | } | |
| 46 |  |

|  |  |
| --- | --- |
| 47 | /\*\* |
| 48 | \* To ensure shutdown | |

|  |  |
| --- | --- |
| 49 | \* |
| 50 | \*/ | |

|  |  |
| --- | --- |
| 51 | @Override |
| 52 | public boolean shutdown(CamelContext context, RouteStartupOrder route, | |

|  |  |  |
| --- | --- | --- |
| 53 | long timeout, TimeUnit timeUnit, boolean abortAfterTimeout) | |
| 54 | throws Exception { |

|  |  |  |
| --- | --- | --- |
| 55 | super.shutdown(context, route, this.timeout, | |
| 56 | this.timeUnit, false); |

|  |  |  |
| --- | --- | --- |
| 57 | return true; | |
| 58 | } |

|  |  |
| --- | --- |
| 59 | } |

**Stop a batch :**

|  |  |
| --- | --- |
| 1 | .process(new Processor() { |
| 2 | public void process(Exchange exchange) throws Exception { | |

|  |  |  |
| --- | --- | --- |
| 3 | context.stop(); | |
| 4 | } |

|  |  |
| --- | --- |
| 5 | }); |

**Calling a method of a bean from a route:**

1. method’s return is always affected to message’s body. For example :
   * public void myMethod(Exchange e) :  
     Will not modify the body
   * public boolean myMethod(Exchange e) :  
     the boolean (or whatever primitive type) will be set in the body
   * public Object myMethod(Exchange e) :  
     the Object will be placed in the body (even if null)
   * public Message myMethod(Exchange e) :  
     the Message will be placed in the body (better avoid this)
   * public List<Object> myMethod(Exchange e) :  
     the list will be set in the body : useful to use with .split(), each object will be sent in a new message
   * public List<Message> myMethod(Exchange e) :  
     the list will be set in the body : a .split() will create a new message for each element (better avoid, see upper)
2. configurable method’s parameters :
   * public void myMethod(Exchange e) :  
     the complete Exchange will be passed
   * public void myMethod(Object o) :  
     Camel will try to convert the body in the required parameter’s class
   * public void myMethod(@Body File o, @Header('myHeader') String myParamHeader) :  
     Camel will inject each parameter as specified

**Exceptions management on routes :**

* in a global way (to be declared before all routes) :

|  |  |
| --- | --- |
| 1 | onException(MyException.class, RuntimeCamelException.class).to(...)... |

* to truly handle Exception and not bubble it in routes (and logs) :

|  |  |
| --- | --- |
| 1 | onException(...).handled(true).to(...)... |

* to continue process in a route after an Exception :

|  |  |
| --- | --- |
| 1 | onException(...).continued(true).to(...)... |

* An exception is ‘handled’ or ‘continued’
* local way (in a route) :

|  |  |
| --- | --- |
| 1 | from(...) |
| 2 | .onException(...).to('manage\_error').log('FAIL !!').end() | |

|  |  |
| --- | --- |
| 3 | .to('continue\_route')... |

For writing file, only the header Exchange.FILE\_NAME is necessary.

**Reorder messages with component .resequence :**

* uses an expression to compute the new order of messages, from a unique Comparable ‘key’ (number, String or custom Comparator)
* two ways :
  + .batch() : batch mode. Waits the reception of ALL the messages befor reorder them. **ATTENTION** : a new thread is created to rethrow messages.
  + .stream() : streaming mode. Uses a gap detection between the messages’ keys to re-send them. It is possible to configure a maximal capacity and a timeout.

**Split the body with a token :**

|  |  |
| --- | --- |
| 1 | .split(body().tokenize('TOKEN')) |

Knowing that the TOKEN will be deleted from content. For example, if receiving a message containing : ‘data1TOKENdata2TOKENdata3’, messages created will be : ‘data1’, ‘data2, ‘data3’. So avoid this when treating XML data, prefer ‘tokenizeXML()’.

**Dynamic access to body’s data :**

* Lightweight ‘script’ language : [Simple Expression Language](http://camel.apache.org/simple.html)
* Read files data : [File Expression Language](http://camel.apache.org/file-language.html)

**Sending mails :**

|  |  |
| --- | --- |
| 1 | from('direct:mail') |
| 2 | .setHeader('To', constant(mailTo)) | |

|  |  |
| --- | --- |
| 3 | .setHeader('From', constant(mailFrom)) |
| 4 | .setHeader('Subject', constant(mailSubject)) | |

|  |  |
| --- | --- |
| 5 | .to('[smtp://](NULL)${user}@${server}:${port}?password=${password}'); |

**With attachment :**

|  |  |  |
| --- | --- | --- |
| 01 | .beanRef(MAIL\_ATTACHER, 'attachLog'); | |
| 02 | //with |

|  |  |
| --- | --- |
| 03 | public class MailAttacher { |
| 04 | public void attachLog(Exchange exc) throws Exception { | |

|  |  |
| --- | --- |
| 05 | File toAttach = ...; |
| 06 | exc.getIn().addAttachment(toAttach.getName(), | |

|  |  |  |
| --- | --- | --- |
| 07 | new DataHandler(new FileDataSource(toAttach))); | |
| 08 | // if needed |

|  |  |  |
| --- | --- | --- |
| 09 | exc.setProperty(Exchange.CHARSET\_NAME, 'UTF-8'); | |
| 10 | } |

|  |  |
| --- | --- |
| 11 | } |

**Useful Exchange’s properties :**

* Exchange.AGGREGATED\_\* : aggregations management
* Exchange.BATCH\_\* : treated messages management
* Exchange.FILE\_\* : File messages management
* Exchange.HTTP\_\* : web requests management
* Exchange.LOOP\_\* : loops management
* Exchange.REDELIVERY\_\* : exceptions management
* Exchange.SPLIT\_\* : splitted contents management

**Loop a route :**

|  |  |
| --- | --- |
| 1 | from('direct:...') |
| 2 | .loop(countExpression) | |

|  |  |  |
| --- | --- | --- |
| 3 | .to('direct:insideLoop') | |
| 4 | .end() |

Where ‘countExpression’ is an Expression used to dynamically compute the loop count (evaluated entering the loop). It is preferable to move the loop’s code in a new route if the process is complex.

**Headers management :**

Message’s headers are defined at its creation. When using a ‘.split()’, all subsequent messages will have the same headers from the original message (so be careful when managing files). In an aggregation, custom headers will have to be managed manually to be preserved in the rest of the route.

**Intercept a message**

and execute a route parallely (to be declared before routes) :

|  |  |
| --- | --- |
| 1 | interceptSendToEndpoint('ENDPOINT\_TO\_INTERSEPT').to(...)... |

|  |
| --- |
| **Polling an empty directory (and send an empty message with null body) :**  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. from("file://temp?sendEmptyMessageWhenIdle=true") |
| **Stop a route :**  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. .process(new Processor() { 2. public void process(Exchange exchange) throws Exception { 3. getContext().stopRoute("ROUTE\_ID"); 4. } 5. }) |
| **Access a property of the object in the body :** admitting the object has a method named "getMydata()" :  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. new ValueBuilder(simple("${body.mydata}")).isEqualTo(...) |
| **Define an aggregator :**  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. .aggregate(simple("${header.id}.substring(0,15)"), 2. genericAggregationStrategy) 3. .completionPredicate(header(Exchange.BATCH\_COMPLETE) 4. .isEqualTo(Boolean.TRUE))  * "${header.id}.substring(0,15)" : flag to differenciate messages (here, the returned string is common to all messages, we aggregate them all) * Exchange.BATCH\_COMPLETE : predicate indicating the end of polling (all files parsed for example) * genericAggregationStrategy : above, an example of an aggregator grouping all messages' contents in a list :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. public class GenericAggregationStrategy implements AggregationStrategy { 2. @SuppressWarnings("unchecked") 3. public Exchange aggregate(Exchange oldExchange, Exchange newExchange) { 4. if (oldExchange == null) { 5. ArrayList<Object> list = new ArrayList<Object>(); 6. list.add(newExchange.getIn().getBody()); 7. newExchange.getIn().setBody(list); 8. return newExchange; 9. } else { 10. Object oldIn = oldExchange.getIn().getBody(); 11. ArrayList<Object> list = null; 12. if(oldIn instanceof ArrayList) { 13. list = (ArrayList<Object>) oldIn; 14. } else { 15. list = new ArrayList<Object>(); 16. list.add(oldIn); 17. } 18. list.add(newExchange.getIn().getBody()); 19. newExchange.getIn().setBody(list); 20. return newExchange; 21. } 22. } 23. } |
| **Manually trigger an aggregation's completion (whatever it is) :**  Send a message with the header Exchange.AGGREGATION\_COMPLETE\_ALL\_GROUPS = true |
| It is possible to do from("bean:..."), knowing that the bean will be polled permanently (like with "file") and re-instanciated each time. |
| **Modify the message's body** on a route, using :  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. .transform(myExpression)   with myExpression :  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. public class MyExpression implements Expression { 2. public <T> T evaluate(Exchange exchange, Class<T> type) { 3. MyBean newData = ...; 4. return exchange.getContext().getTypeConverter() 5. .convertTo(type, newData); 6. } 7. } |
| **Using JaxB :**   * on a route :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   * 1. .[un]marshal().jaxb("my.business\_classes.package") * with a configurable DataFormat :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   * 1. .[un]marshal(jaxbDataFormat)   with :  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. // indicate to Jaxb to not write XML prolog : 2. JaxbDataFormat jaxbDataFormat = 3. new JaxbDataFormat("my.business\_classes.package"); 4. jaxb.setFragment(true); |
| **General concepts for threads management :**   * a from(...) = a thread * except for from("direct:...") wich creates a "named route" with a unique identifier only callable by another route (in the same thread than the caller). * The component .resequence().batch() creates a new thread to rethrow the messages. |
| **Define a shutdown strategy :**  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. getContext().setShutdownStrategy(new MyShutdownStrategy(getContext()));   With :  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. public class MyShutdownStrategy extends DefaultShutdownStrategy { 2. protected CamelContext camelContext; 3. private long timeout = 1; 4. private TimeUnit timeUnit = TimeUnit.SECONDS; 5. public SpiralShutdownStrategy(CamelContext camelContext) { 6. this.camelContext = camelContext; 7. } 9. @Override 10. public long getTimeout() { 11. return this.timeout; 12. } 14. @Override 15. public TimeUnit getTimeUnit() { 16. return this.timeUnit; 17. } 19. @Override 20. public CamelContext getCamelContext() { 21. return this.camelContext; 22. } 24. /\*\* 25. \* To ensure shutdown 26. \* 27. \*/ 28. @Override 29. public void suspend(CamelContext context, 30. List<RouteStartupOrder> routes) throws Exception { 31. doShutdown(context, routes, getTimeout(), 32. getTimeUnit(), false, false, false); 33. } 35. /\*\* 36. \* To ensure shutdown 37. \* 38. \*/ 39. @Override 40. public void shutdown(CamelContext context, 41. List<RouteStartupOrder> routes, long timeout, 42. TimeUnit timeUnit) throws Exception { 43. doShutdown(context, routes, this.timeout, 44. this.timeUnit, false, false, false); 45. } 47. /\*\* 48. \* To ensure shutdown 49. \* 50. \*/ 51. @Override 52. public boolean shutdown(CamelContext context, RouteStartupOrder route, 53. long timeout, TimeUnit timeUnit, boolean abortAfterTimeout) 54. throws Exception { 55. super.shutdown(context, route, this.timeout, 56. this.timeUnit, false); 57. return true; 58. } 59. } |
| **Stop a batch :**  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. .process(new Processor() { 2. public void process(Exchange exchange) throws Exception { 3. context.stop(); 4. } 5. }); |
| **Calling a method of a bean from a route** :   1. method's return is always affected to message's body. For example :    * public void myMethod(Exchange e) : Will not modify the body    * public boolean myMethod(Exchange e) : the boolean (or whatever primitive type) will be set in the body    * public Object myMethod(Exchange e) : the Object will be placed in the body (even if null)    * public Message myMethod(Exchange e) : the Message will be placed in the body (better avoid this)    * public List<Object> myMethod(Exchange e) : the list will be set in the body : useful to use with .split(), each object will be sent in a new message    * public List<Message> myMethod(Exchange e) : the list will be set in the body : a .split() will create a new message for each element (better avoid, see upper) 2. configurable method's parameters :    * public void myMethod(Exchange e) : the complete Exchange will be passed    * public void myMethod(Object o) : Camel will try to convert the body in the required parameter's class    * public void myMethod(@Body File o, @Header("myHeader") String myParamHeader) : Camel will inject each parameter as specified |
| **Exceptions management on routes :**   * in a global way (to be declared before all routes) :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   * 1. onException(MyException.class, RuntimeCamelException.class).to(...)... * to truly handle Exception and not bubble it in routes (and logs) :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   * 1. onException(...).handled(true).to(...)... * to continue process in a route after an Exception :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   * 1. onException(...).continued(true).to(...)... * An exception is "handled" or "continued" * local way (in a route) :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   * 1. from(...)   2. .onException(...).to("manage\_error").log("FAIL !!").end()   3. .to("continue\_route")... |
| For writing file, only the header Exchange.FILE\_NAME is necessary. |
| **Reorder messages with component .resequence :**   * uses an expression to compute the new order of messages, from a unique Comparable "key" (number, String or custom Comparator) * two ways :   + .batch() : batch mode. Waits the reception of ALL the messages befor reorder them. **ATTENTION** : a new thread is created to rethrow messages.   + .stream() : streaming mode. Uses a gap detection between the messages' keys to re-send them. It is possible to configure a maximal capacity and a timeout. |
| **Split the body with a token :**  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. .split(body().tokenize("TOKEN"))   Knowing that the TOKEN will be deleted from content. For example, if receiving a message containing : "data1TOKENdata2TOKENdata3", messages created will be : "data1", "data2, "data3".  So avoid this when treating XML data, prefer "tokenizeXML()". |
| **Dynamic access to body's data :**   * Lightweight "script" language : [Simple Expression Language](http://camel.apache.org/simple.html) * Read files data : [File Expression Language](http://camel.apache.org/file-language.html) |
| **Sending mails :**  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. from("direct:mail") 2. .setHeader("To", constant(mailTo)) 3. .setHeader("From", constant(mailFrom)) 4. .setHeader("Subject", constant(mailSubject)) 5. .to("smtp://${user}@${server}:${port}?password=${password}");   **With attachment :**  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. .beanRef(MAIL\_ATTACHER, "attachLog"); 2. //with 3. public class MailAttacher { 4. public void attachLog(Exchange exc) throws Exception { 5. File toAttach = ...; 6. exc.getIn().addAttachment(toAttach.getName(), 7. new DataHandler(new FileDataSource(toAttach))); 8. // if needed 9. exc.setProperty(Exchange.CHARSET\_NAME, "UTF-8"); 10. } 11. } |
| **Useful Exchange's properties :**   * Exchange.AGGREGATED\_\* : aggregations management * Exchange.BATCH\_\* : treated messages management * Exchange.FILE\_\* : File messages management * Exchange.HTTP\_\* : web requests management * Exchange.LOOP\_\* : loops management * Exchange.REDELIVERY\_\* : exceptions management * Exchange.SPLIT\_\* : splitted contents management * Exchange.EXCEPTION\_CAUGHT : thrown exception on route |
| **Loop a route :**  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. from("direct:...") 2. .loop(countExpression) 3. .to("direct:insideLoop") 4. .end()   Where "countExpression" is an Expression used to dynamically compute the loop count (evaluated entering the loop)  It is preferable to move the loop's code in a new route if the process is complex. |
| **Headers management :**  Message's headers are defined at its creation. When using a ".split()", all subsequent messages will have the same headers from the original message (so be careful when managing files). In an aggregation, custom headers will have to be managed manually to be preserved in the rest of the route. |
| **Intercept a message** and execute a route parallely (to be declared before routes) :  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. interceptSendToEndpoint("ENDPOINT\_TO\_INTERSEPT").to(...)... |
| **Send an exchange from a Java class to a route :**   1. Create a ProducerTemplate from Camel context :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   * 1. ProducerTemplate template = camelRouteBuilder   2. .getContext().createProducerTemplate();  1. Define its target endpoint :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   * 1. template.setDefaultEndpointUri("direct:start");  1. Create a message and Exchange from template :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   * 1. Message message = new DefaultMessage();   2. message.setBody("...");   3. message.setHeader(key,value);   5. Exchange exchange = template.getDefaultEndpoint().createExchange();   6. exchange.setIn(message);  1. Send message :   [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   * 1. template.send(exchange); |
| **Definie a maximum inflight messages** with a route policy :  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. ThrottlingInflightRoutePolicy policy = new ThrottlingInflightRoutePolicy(); 2. policy.setMaxInflightExchanges(999); 3. policy.setScope(ThrottlingInflightRoutePolicy.ThrottlingScope.Route); 5. from("seda:route").routePolicy(policy) |
| **Best pratices** to combine Split and aggregator :  It is possible to use :  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. .split(body()) 2. (...) 3. .aggregate(simple("${header.id}.substring(0,15)"), simpleAggregationStrategy) 4. .completionPredicate(header(Exchange.SPLIT\_COMPLETE).isEqualTo(Boolean.TRUE)) 5. (... end of route ...)   But with an empty list (null body or list size to 0), split behavior must be confusing and might shortcut the end of the route. Safer use is :  [view plainprint?](http://developpef.blogspot.in/2013/01/camel-cheatsheet.html)   1. .split(body(), simpleAggregationStrategy) 2. (...) 3. .end()   The end() instruction clearly isolates the split instructions block and simpleAggregationStrategy is still triggered at SPLIT\_COMPLETE |

To be continued, don't hesitate to participate!

**Stream Component**

The **stream:** component provides access to the System.in, System.out and System.err streams as well as allowing streaming of file and URL.

Maven users will need to add the following dependency to their pom.xml for this component:

|  |
| --- |
| <dependency>      <groupId>org.apache.camel</groupId>      <artifactId>camel-stream</artifactId>      <version>x.x.x</version>      <!-- use the same version as your Camel core version -->  </dependency> |

**URI format**

|  |
| --- |
| stream:in[?options]  stream:out[?options]  stream:err[?options]  stream:header[?options] |

In addition, the file and url endpoint URIs are supported:

|  |
| --- |
| stream:file?fileName=/foo/bar.txt  stream:url[?options] |

If the stream:header URI is specified, the stream header is used to find the stream to write to. This option is available only for stream producers (that is, it cannot appear in from()).

You can append query options to the URI in the following format, ?option=value&option=value&...

**Options**

|  |  |  |
| --- | --- | --- |
| **Name** | **Default Value** | **Description** |
| delay | 0 | Initial delay in milliseconds before consuming or producing the stream. |
| encoding | *JVM Default* | You can configure the encoding (is a [charset name](http://java.sun.com/j2se/1.5.0/docs/api/java/nio/charset/Charset.html)) to use text-based streams (for example, message body is a String object). If not provided, Camel uses the [JVM default Charset](http://java.sun.com/j2se/1.5.0/docs/api/java/nio/charset/Charset.html#defaultCharset%28%29). |
| promptMessage | null | Message prompt to use when reading from stream:in; for example, you could set this to Enter a command: |
| promptDelay | 0 | Optional delay in milliseconds before showing the message prompt. |
| initialPromptDelay | 2000 | Initial delay in milliseconds before showing the message prompt. This delay occurs only once. Can be used during system startup to avoid message prompts being written while other logging is done to the system out. |
| fileName | null | When using the stream:file URI format, this option specifies the filename to stream to/from. |
| url | null | When using the stream:url URI format, this option specifies the URL to stream to/from. The input/output stream will be opened using the [JDK URLConnection](http://docs.oracle.com/javase/6/docs/api/java/net/URLConnection.html) facility. |
| scanStream | false | To be used for continuously reading a stream such as the unix tail command.  **Camel 2.4 to Camel 2.6:** will retry opening the file if it is overwritten, somewhat like tail --retry |
| retry | false | **Camel 2.7:** will retry opening the file if it's overwritten, somewhat like tail --retry |
| scanStreamDelay | 0 | Delay in milliseconds between read attempts when using scanStream. |
| groupLines | 0 | **Camel 2.5:** To group X number of lines in the consumer. For example to group 10 lines and therefore only spit out an [Exchange](http://camel.apache.org/exchange.html) with 10 lines, instead of 1 [Exchange](http://camel.apache.org/exchange.html) per line. |
| autoCloseCount | 0 | **Camel 2.10.0:** (2.9.3 and 2.8.6) Number of messages to process before closing stream on Producer side. Never close stream by default (only when Producer is stopped). If more messages are sent, the stream is reopened for another autoCloseCount batch. |
| closeOnDone | false | **Camel 2.11.0:** This option is used in combination with [Splitter](http://camel.apache.org/splitter.html) and streaming to the same file. The idea is to keep the stream open and only close when the [Splitter](http://camel.apache.org/splitter.html) is done, to improve performance. Mind this requires that you only stream to the same file, and not 2 or more files, and that the last split message that carries the information that its the last, is routed to the stream endpoint so it gets the signal to close. |

**Message content**

The **stream:** component supports either String or byte[] for writing to streams. Just add either String or byte[] content to the message.in.body. Messages sent to the **stream:** producer in binary mode are not followed by the newline character (as opposed to the String messages). Message with null body will not be appended to the output stream.  
The special stream:header URI is used for custom output streams. Just add a java.io.OutputStream object to message.in.header in the key header.  
See samples for an example.

**Samples**

In the following sample we route messages from the direct:in endpoint to the System.out stream:

|  |
| --- |
| // Route messages to the standard output.  from("direct:in").to("stream:out");    // Send String payload to the standard output.  // Message will be followed by the newline.  template.sendBody("direct:in", "Hello Text World");    // Send byte[] payload to the standard output.  // No newline will be added after the message.  template.sendBody("direct:in", "Hello Bytes World".getBytes()); |

The following sample demonstrates how the header type can be used to determine which stream to use. In the sample we use our own output stream, MyOutputStream.

|  |
| --- |
| private OutputStream mystream = new MyOutputStream();  private StringBuilder sb = new StringBuilder();    @Test  public void testStringContent() {      template.sendBody("direct:in", "Hello");      // StreamProducer appends \n in text mode      assertEquals("Hello\n", sb.toString());  }    @Test  public void testBinaryContent() {      template.sendBody("direct:in", "Hello".getBytes());      // StreamProducer is in binary mode so no \n is appended      assertEquals("Hello", sb.toString());  }    protected RouteBuilder createRouteBuilder() {      return new RouteBuilder() {          public void configure() {              from("direct:in").setHeader("stream", constant(mystream)).                  to("stream:header");          }      };  }    private class MyOutputStream extends OutputStream {        public void write(int b) throws IOException {          sb.append((char)b);      }  } |

The following sample demonstrates how to continuously read a file stream (analogous to the UNIX tail command):

|  |
| --- |
| from("stream:file?fileName=/server/logs/server.log&scanStream=true&scanStreamDelay=1000").to("bean:logService?method=parseLogLine"); |

One gotcha with scanStream (pre Camel 2.7) or scanStream + retry is the file will be re-opened and scanned with each iteration of scanStreamDelay. Until NIO2 is available we cannot reliably detect when a file is deleted/recreated.

**Console Example**

**Available as of Camel 2.10**

This example is located in the [Camel distribution](http://camel.apache.org/download.html) at examples/camel-example-console.

This is a beginner's example that demonstrates how to get started with Apache Camel.

In this example we integrate with the console using the [Stream](http://camel.apache.org/stream.html) component. The example is interactive - it reads input from the console, and then transforms the input to upper case and prints it back to the console.

This is implemented with a Camel route defined in the Spring XML markup shown below:

|  |
| --- |
| <!-- camelContext is the Camel runtime, where we can host Camel routes -->  <camelContext xmlns="[http://camel.apache.org/schema/spring"](http://camel.apache.org/schema/spring%22)>    <route>      <!-- read input from the console using the stream component -->      <from uri="stream:in?promptMessage=Enter something: "/>      <!-- transform the input to upper case using the simple language -->      <!-- you can also use other languages such as groovy, ognl, mvel, javascript etc. -->      <transform>        <simple>${body.toUpperCase()}</simple>      </transform>      <!-- and then print to the console -->      <to uri="stream:out"/>    </route>  </camelContext> |

This example can be launched from the command line using Maven:

|  |
| --- |
| mvn compile exec:java |

In the console you can enter a message and press <ENTER>. Camel responds by echoing the input message in upper case, as shown below:

|  |
| --- |
| [onsole.CamelConsoleMain.main()] SpringCamelContext             INFO  Apache Camel 2.10 (CamelContext: camel-1) started in 0.455 seconds  Enter something: camel rocks  CAMEL ROCKS  Enter something: and we have fun  AND WE HAVE FUN  Enter something: |

To stop the example, strike Control+C

You can also run this example from your editor. For example, from Eclipse you can import this project using: File → Import … → Existing Maven Project, and select pom.xml from the examples\camel-example-console directory.

Next, navigate to the org.apache.camel.example.console.CamelConsoleMain class, right-click, and select Run As → Java Application.

Click on the screenshot below, to make it bigger.

