#47

Get More Refcardz! Visit refcardz.com

The Top Twelve Integration Patterns for Apache Camel

DZone Refcardz

brought to you by...
FuseSource

The Top Twelve Integration Patterns for Apache Camel

Update for Apache Camel

By Claus Ibsen

ABOUT ENTERPRISE INTEGRATION PATTERNS

Integration is a complex problem. To help deal with the complexity of integration problems, the Enterprise Integration Patterns (EIP) have become the standard way to describe, document and implement complex integration problems. Hohpe & Woolf's book the Enterprise Integration Patterns has become the bible in the integration space – essential reading for any integration professional.

Apache Camel is an open-source project for implementing the EIP easily in a few lines of Java code or XML configuration. This Refcard guides you through the most common Enterprise Integration Patterns and gives you examples of how to implement them either in Java code or using XML. This Refcard is targeted at software developers and enterprise architects, but anyone in the integration space can benefit as well.

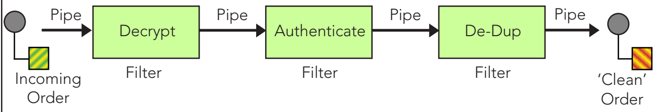
ABOUT APACHE CAMEL

Apache Camel is a powerful open-source integration platform based on Enterprise Integration Patterns (EIP) with powerful bean integration. Camel lets you implement EIP routing using Camels intuitive Domain Specific Language (DSL) based on Java (aka fluent builder) or XML. Camel uses URI for endpoint resolution so it's very easy to work with any kind of transport such as JMS, HTTP, SOAP, REST, File, FTP, TCP, XMPP, JBI, SMTP, and many others. Camel also provides data formats for various popular formats such as CSV, EDI, FIX, HL7, JAXB, and JSon, etc. Camel is an integration API that can be embedded in any server of choice such as Apache ServiceMix, ActiveMQ, Tomcat, Jetty, JEE Application Server, standalone, or in the cloud. Camel is OSGi compliant, allowing you to host your Camel bundles in an OSGi container such as Apache ServiceMix. Camel's bean integration let you define loose coupling allowing you to fully separate your business logic from the integration logic. Camel is based on a modular architecture allowing you to plug in your own component or data format, so they seamlessly blend in with existing modules. Camel provides a test kit for unit and integration testing with strong mock and assertion capabilities.

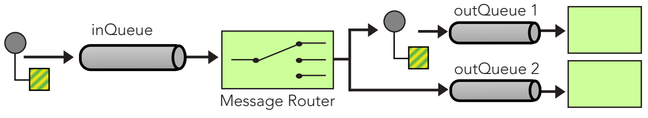
ESSENTIAL PATTERNS

This group consists of the most essential patterns that anyone working with integration must know.

Pipes and Filters

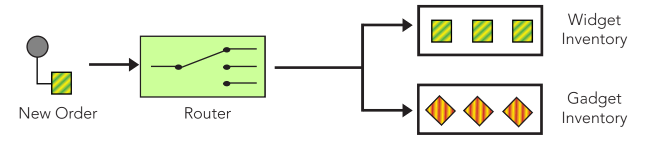
<table>
<tr><td></td><td>How can we perform complex processing on a message while maintaining independence and flexibility?</td></tr>
<tr><td colspan="2">
</td></tr>
<tr><td><b>Problem</b></td><td>A single event often triggers a sequence of processing steps.</td></tr>
<tr><td><b>Solution</b></td><td>Use Pipes and Filters to divide a larger processing steps (filters) that are connected by channels (pipes).</td></tr>
<tr><td><b>Camel</b></td><td>Camel supports Pipes and Filters using the <b>pipeline</b> node.</td></tr>
<tr><td><b>Java DSL</b></td><td><pre>from("jms:queue:order:in").pipeline("direct:transformOrd
er", "direct:validateOrder", "jms:queue:order:process");</pre>
Where jms represents the JMS component used for consuming JMS messages on the JMS broker. Direct is used for combining endpoints in a synchronous fashion, allowing you to divide routes into sub routes and/or reuse common routes.
<b>Tip:</b> Pipeline is the default mode of operation when you specify multiple outputs, so it can be omitted and replaced with the more common node:
<pre>from("jms:queue:order:in").to("direct:transformOrder",
"direct:validateOrder", "jms:queue:order:process");</pre>
<b>TIP:</b> You can also separate each step as individual <b>to</b> nodes:
<pre>from("jms:queue:order:in")
    .to("direct:transformOrder")
    .to("direct:validateOrder")
    .to("jms:queue:order:process");</pre></td></tr>
<tr><td><b>XML DSL</b></td><td><pre>&lt;route>
  <from uri="jms:queue:order:in"/>
  <pipeline>
    <to uri="direct:transformOrder"/>
    <to uri="direct:validateOrder"/>
    <to uri="jms:queue:order:process"/>
  </pipeline>
</route>
<route>
  <from uri="jms:queue:order:in"/>
  <to uri="direct:transformOrder"/>
  <to uri="direct:validateOrder"/>
  <to uri="jms:queue:order:process"/>
</route></pre></td></tr>
</table>

Message Router

<table>
<tr><td></td><td>How can you deouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?</td></tr>
<tr><td colspan="2">
</td></tr>
<tr><td><b>Problem</b></td><td>Pipes and Filters route each message in the same processing steps. How can we route messages differently?</td></tr>
<tr><td><b>Solution</b></td><td>Filter using predicates to choose the right output destination.</td></tr>
<tr><td><b>Camel</b></td><td>Camel supports Message Router using the <b>choice</b> node. For more details see the Content-Based router pattern.</td></tr>
</table>

Content-Based Router

How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?



Problem

How do we ensure a Message is sent to the correct recipient based on information from its content?

Solution

Use a Content-Based Router to route each message to the correct recipient based on the message content.

Camel

Camel has extensive support for Content-Based Routing. Camel supports content based routing based on choice, filter, or any other expression.

Java DSL

```
from(“jms:order.process”)
  .choice()
    .when(header(“type”).isEqualTo(“widget”))
      .to(“jms:order.widget”)
    .when(header(“type”).isEqualTo(“gadget”))
      .to(“jms:order.gadget”)
    .otherwise()
       .to(“jms:order.other”);
```

XML DSL

```
<route>
  <from uri=”jms:order.process”/>
  <choice>
    <when>
      <simple>${header.type} == ‘widget’</simple>
      <to uri=”jms:order.widget”/>
    </when>
    <when>
      <simple>${header.type} == ‘gadget’</simple>
      <to uri=”jms:order.gadget”/>
    </when>
    <otherwise>
      <to uri=”jms:order.other”/>
    </otherwise>
  </choice>
</route>
```

TIP: In XML DSL you cannot invoke code, as opposed to the Java DSL. To express the predicates for the choices we need to use a language. We will use simple language that uses a simple expression parser that supports a limited set of operators. You can use any of the more powerful languages supported in Camel such as: JavaScript, Groovy, Unified EL and many others.

TIP: You can also use a method call to invoke a method on a bean to evaluate the predicate. Lets try that:

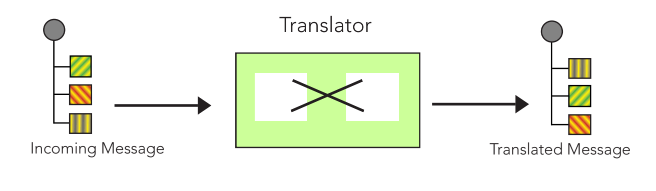
```
<when>
  <method bean=”myBean” method=”isGadget”/>
  ...
</when>
<bean id=”myBean” class=”com.mycomapany.MyBean”/>

public boolean isGadget(@Header(name = “type”) String
type) {
    return type.equals(“Gadget”);
}
```

Notice how we use Bean Parameter Binding to instruct Camel to invoke this method and pass in the type header as the String parameter. This allows your code to be fully decoupled from any Camel API so its easy to read, write and unit test.

Message Translator

How can systems using different data formats communicate with each other using messaging?



Problem

Each application uses its own data format, so we need to translate the message into the data format the application supports.

Solution

Use a special filter, a message translator, between filters or applications to translate one data format into another.

Camel

Camel supports the message translator using the **processor**, **bean** or **transform** nodes.

TIP: Camel routes the message as a chain of **processor** nodes.

Java DSL

```
public class OrderTransformProcessor
        implements Processor {
    public void process(Exchange exchange)
        throws Exception {
        // do message translation here
    }
 }

from(“direct:transformOrder”)
    .process(new OrderTransformProcessor());
```

Bean

Instead of the processor, we can use Bean (POJO). An advantage of using a Bean over Processor is the fact that we do not have to implement or use any Camel specific interfaces or types. This allows you to fully decouple your beans from Camel.

```
public class OrderTransformerBean {
    public StringtransformOrder(String body) {
        // do message translation here
    }
}

Object transformer = new OrderTransformerBean();
from(“direct:transformOrder”).bean(transformer);
```

TIP: Camel can create an instance of the bean automatically; you can just refer to the class type.

```
from(“direct:transformOrder”)
    .bean(OrderTransformerBean.class);
```

TIP: Camel will try to figure out which method to invoke on the bean in case there are multiple methods. In case of ambiguity, you can specify which methods to invoke by the method parameter:

```
from(“direct:transformOrder”)
    .bean(OrderTransformerBean.class, “transformOrder”);
```

Transform

Transform is a particular processor allowing you to set a response to be returned to the original caller. We use transform to return a constant ACK response to the TCP listener after we have copied the message to the JMS queue. Notice we use a constant to build an “ACK” string as response.

```
from(“mina:tcp://localhost:8888?textline=true”)
    .to(“jms:queue:order:in”)
    .transform(constant(“ACK”));
```

XML DSL

Processor

```
<route>
  <from uri=”direct:transformOrder”/>
  <process ref=”transformer”/>
</route>

<bean id=”transformer” class=”com.mycompany.
OrderTransformProcessor”/>
```

In XML DSL, Camel will look up the processor or POJO/Bean in the registry based on the id of the bean.

Bean

```
<route>
  <from uri=”direct:transformOrder”/>
  <bean ref=”transformer”/>
</route>

<bean id=”tramsformer”
      class=”com.mycompany.OrderTransformBean”/>
```

Transform

```
<route>
  <from uri=”mina:tcp://localhost:8888?textline=true”/>
  <to uri=”jms:queue:order:in”/>
  <transform>
    <constant>ACK</constant>
  </transform>
</route>
```

Annotation DSL

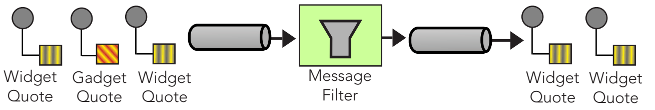
You can also use the **@Consume** annotation for transformations. For example, in the method below we consume from a JMS queue and do the transformation in regular Java code. Notice that the input and output parameters of the method is String. Camel will automatically coerce the payload to the expected type defined by the method. Since this is a JMS example, the response will be sent back to the JMS reply-to destination.

```
@Consume(uri=”jms:queue:order:transform”)
public String transformOrder(String body) {
    // do message translation
}
```

TIP: You can use Bean Parameter Binding to help Camel coerce the Message into the method parameters. For instance, you can use **@Body**, **@Headers** parameter annotations to bind parameters to the body and headers.

Message Filter

How can a component avoid receiving unwanted messages?



Problem

How do you discard unwanted messages?

Solution

Use a special kind of Message Router, a Message Filter, to eliminate undesired messages from a channel based on a set of criteria.

Camel

Camel has support for Message Filter using the filter method. The filter evaluates a predicate whether its true or false; only allowing the true condition to pass the filter, where as the false condition will silently be ignored.

Java DSL

We want to discard any test messages so we only route non-test messages to the order queue.

```
from(“jms:inbox”)
  .filter(header(“test”).isNotEqualTo(“true”))
    .to(“jms:order”);
```

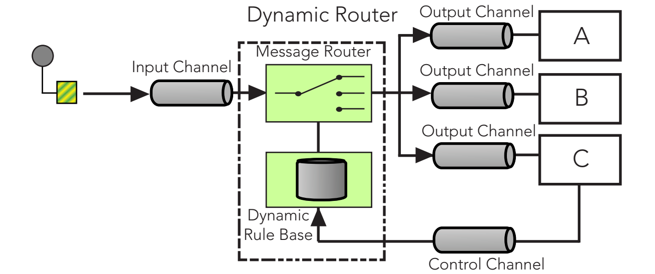
XML DSL

In the XML DSL we use the built-in expression language (simple) to define the predicate to be used by the filter.

```
<route>
  <from uri=”jms:inbox”/>
  <filter>
    <simple>${header.test} == false</simple>
    <to uri=”jms:order”/>
  </filter>
</route>
```

Dynamic Router

How can you avoid the dependency of the router on all possible destinations while maintaining its efficiency?



Problem

How can we route messages based on a dynamic list of destinations?

Solution

Use a Dynamic Router, a router that can self-configure based on special configuration messages from participating destinations.

Camel

Camel has support for Dynamic Router using the dynamicRouter method. An expression must be provided to determine where the message should be routed next. After the message has been routed Camel will re-evaluate the expression to compute where the message should go next. It will keep doing this until the expression returns null to indicate the end.

Java DSL

We use a bean as the expression to compute where the message should be routed.

```
public class MyRouter {
  public String whereToGo(String body) {
    // query a data store to find where we should go next.
Return null to indicate end.
  }
}
```

We can then use the bean in the dynamicRouter in the Camel route:

```
from(“jms:queue:order”)
  .dynamicRouter(bean(new MyRouter()));
```

XML DSL

In XML DSL we have to define the bean as a regular spring bean.

```
<bean id=”router” class=”com.foo.MyRouter”/>
```

Which we then can refer to in the <dynamicRouter> tag.

```
<route>
  <from uri=”jms:queue:order”/>
  <dynamicRouter>
    <method ref=”router”/>
  </dynamicRouter>
</route>
```

Annotation DSL

You can use @DynamicRouter annotation in a bean to turn it into a dynamic router.

```
public class MyRouterBean {
  @DynamicRouter
  public String route(Exchange exchange) {
   // query a data store to find where we should go next.
Return null to indicate end.
    }
}
```

Annotation DSL, continued

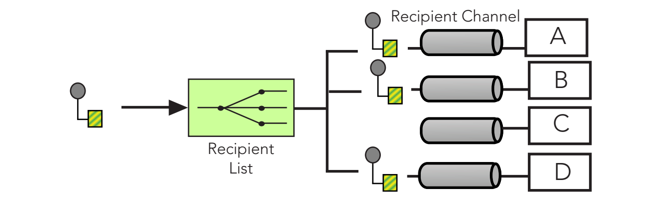
Pay attention to the Camel route as you must invoke the bean as if it were a regular bean instead.

```
from(“jms:queue:order”)
  .bean(new MyRouterBean());
```

TIP: You can use @Body, @Header, and @Headers annotations to bind parameters to the message body and headers in the method signature of the route method on the bean.

Recipient List

How do we route a message to a list of statically or dynamically specified recipients?



Problem

How can we route messages based on a static or dynamic list of destinations?

Solution

Define a channel for each recipient. Then use a Recipient List to inspect an incoming message, determine the list of desired recipients and forward the message to all channels associated with the recipients in the list.

Camel

Camel supports the static Recipient List using the **multicast** node, and the dynamic Recipient List using the **recipientList** node.

Java DSL

Static

In this route, we route to a static list of two recipients that will receive a copy of the same message simultaneously.

```
from(“jms:queue:inbox”)
    .multicast().to(“file://backup”, “seda:inbox”);
```

Dynamic

In this route, we route to a dynamic list of recipients defined in the message header [mails] containing a list of recipients as endpoint URLs. The bean processMails is used to add the header[mails] to the message.

```
from(“seda:confirmMails”).beanRef(processMails)
    .recipientList(“destinations”);
```

And in the process mails bean we use **@Headers** Bean Parameter Binding to provide a `java.util.Map` to store the recipients.

```
public void confirm(@Headers Map headers, @Body String
body} {
    String[] recipients = ...
    headers.put(“destinations”, recipients);
}
```

XML DSL

Static

```
<route>
  <from uri=”jms:queue:inbox/>
  <multicast>
    <to uri=”file://backup”/>
    <to uri=”seda:inbox”/>
  </multicast>
</route>
```

Dynamic

In this example, we invoke a method call on a Bean to provide the dynamic list of recipients.

```
<route>
  <from uri=”jms:queue:inbox/>
  <recipientList>
    <method bean=”myDynamicRouter” method=”route”/>
  </recipientList>
</route>
```

```
<bean id=”myDynamicRouter”
      class=”com.mycompany.MyDynamicRouter”/>
```

```
public class myDynamicRouter {
    public String[] route(String body) {
      return new String[] { “file://backup”, .... }
    }
}
```

Annotation DSL

In the CustomerService, class we annoate the `whereTo` method with **@RecipientList** and return a single destination based on the customer id. Notice the flexibility of Camel as it can adapt accordingly to how you define what your methods are returning: a single element, a list, an iterator, etc.

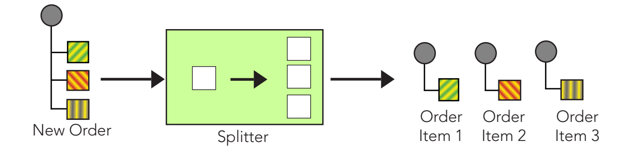
```
public class CustomerService {
    @RecipientList
    public String whereTo(@Header(“customerId”) id) {
      return “jms:queue:customer:” + id;
    }
}
```

And then we can route to the bean and it will act as a dynamic recipient list.

```
from(“jms:queue:inbox”)
    .bean(CustomerService.class, “whereTo”);
```

Splitter

How can a component avoid receiving unwanted messages?



Problem	How can we split a single message into pieces to be routed individually?
Solution	Use a Splitter to break out the composite message into a series of individual messages, each containing data related to one item.
Camel	Camel has support for Splitter using the **split** node.

Java DSL

In this route, we consume files from the inbox folder. Each file is then split into a new message. We use a **tokenizer** to split the file content line by line based on line breaks.

```
from(“file://inbox”)
    .split(body().tokenize(“\n”))
    .to(“seda:orderLines”);
```

TIP: Camel also supports splitting streams using the streaming node. We can split the stream by using a comma:

```
.split(body().tokenize(“,”)).streaming().
to(“seda:parts”);
```

TIP: In the routes above each individual split message will be executed in sequence. Camel also supports parallel execution using the parallelProcessing node.

```
.split(body().tokenize(“,”)).streaming()
    .parallelProcessing().to(“seda:parts”);
```

XML DSL

In this route, we use XPath to split XML payloads received on the JMS order queue.

```
<route>
  <from uri=”jms:queue:order”/>
  <split>
    <xpath>/invoice/lineItems</xpath>
    <to uri=”seda:processOrderLine”/>
  </split>
</route>
```

And in this route we split the messages using a regular expression:

```
<route>
  <from uri=”jms:queue:order”/>
  <split>
    <tokenizer token=”([A-Z|0-9]*);” regex=”true”/>
    <to uri=”seda:processOrderLine”/>
  </split>
</route>
```

TIP: Split evaluates an `org.apahce.camel.Expression` to provide something that is iterable to produce each individual new message. This allows you to provide any kind of expression such as a Bean invoked as a method call.

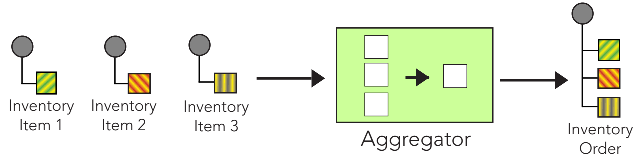
```
  <split>
    <method bean=”mySplitter” method=”splitMe”/>
    <to uri=”seda:processOrderLine”/>
  </split>

  <bean id=”mySplitter” class=”com.mycompany.
MySplitter”/>

public List splitMe(String body) {
    // split using java code and return a List
    List parts = ...
    return parts;
}
```

Aggregator

How do we combine the results of individual, but related messages so that they can be processed as a whole?



Problem	How do we combine multiple messages into a single combined message?
Solution	Use a stateful filter, an Aggregator, to collect and store individual messages until it receives a complete set of related messages to be published.
Camel	Camel has support for the Aggregator using the aggregate method. A correlation expression is used to determine which messages are related. An aggregation strategy is used to combine aggregated messages into the outgoing message. Camel's aggregator also supports a completion predicate allowing you to signal when the aggregation is complete.

Java DSL

Stock quote example
We want to update a website 5th second with the latest stock quotes. The quotes are received on a JMS topic. As we can receive multiple quotes for the same stock within this time period we only want to keep the last one as its the most up to date. We can do this with the aggregator:

```
from(“jms:topic:stock:quote”)
  .aggregate()
    .xpath(“/quote/@symbol”)
    .completionInterval(5000)
    .to(“direct:quotes”);
```

As the correlation expression we use XPath to fetch the stock symbol from the message body. As the aggregation strategy we use the default provided by Camel that picks the latest message, and thus also the most up to date. To trigger the outgoing messages to be published we use a completion interval set to 5 seconds.

Loan broker example
We aggregate responses from various banks for their quote for a given loan request. We want to pick the bank with the best quote (the cheapest loan), therefore we need to base our aggregation strategy to pick the best quote.

```
from(“jms:topic:loan:quote”)
  .aggregate()
    .header(“loanId”)
    .aggregationStrategy(bestQuote)
    .completionSize(3)
      .to(“direct:bestLoanQuote”);
```

We wish to trigger completion when we have received 3 quotes to pick the best among. The following shows the code snippet for the aggregation strategy we must implement to pick the best quote:

```
 public class BestQuoteStrategy implements
AggregationStrategy {
  public Exchange aggregate(Exchange oldExchange, Exchange
newExchange) {
    double oldQuote = oldExchange.getIn().getBody(Double.
class);
    double newQuote = newExchange.getIn().getBody(Double.
class);
    // return the “winner” that has the lowest quote
    return newQuote < oldQuote ? newExchange : oldExchange;
  }
}
```

XML DZL

Stock quote example

```
<route>
  <from uri=”jms:topic:stock:quote”/>
  <aggregate completionInterval=”5000”>
    <correlationExpression>
      <xpath>/quote/@symbol</xpath>
    </correlationExpression>
    <to uri=”direct:quotes”/>
  </aggregate>
</route>
```

Loan Broker Example

```
<bean id=”bestQuote” class=”com.mycompany.
BestQuoteStrategy”/>
<route>
  <from uri=”jms:topic:loan:qoute”/>
  <aggregate strategyRef=”bestQuote” completionSize=”3”>
    <correlationExpression>
      <header>loanId</header>
    </correlationExpression>
    <to uri=”seda:bestLoanQuote”/>
  </aggregate>
</route>
```

TIP: The aggregate supports 5 different types of completions such as based on timeout, inactivity, a predicate, or size. You can use configure multiple completions such as a timeout and a size.

TIP: If the completed predicate is more complex we can use a method call to invoke a bean so we can do the evaluation in pure Java code:

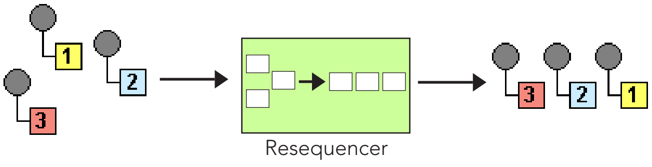
```
<bean id=”quoteService” class=”com.foo.QuoteService”/>
public boolean isComplete(String body) {
  return body.equals(“STOP”);
}
<completionPredicate>
  <method bean=”quoteService” method=”isComplete”/>
</completionPredicate>
```

Which can be even simpler using the Simple expression language:

```
<completionPredicate>
  <simple>${body} == STOP</simple>
</completionPredicate>
```

Resequencer

How can we get a stream of related but out-of-sequence messages back into the correct order?



Problem

How do we ensure ordering of messages?

Solution

Use a stateful filter, a Resequencer, to collect and reorder messages so that they can be published in a specified order.

Camel

Camel has support for the Resequencer using the resequence node. Camel uses a stateful batch processor that is capable of reordering related messages. Camel supports two resequencing algorithms:

batch: collects messages into a batch, sorts the messages and publishes the messages.
stream: reorders, continuously, message streams based on detection of gaps between messages.

Batch is similar to the aggregator but with sorting. Stream is the traditional Resequencer pattern with gap detection. Stream requires usage of number (longs) as sequencer numbers, enforced by the gap detection, as it must be able to compute if gaps exist. A gap is detected if a number in a series is missing, (e.g. 3, 4, 6 with number 5 missing). Camel will back off the messages until number 5 arrives.

Java DSL

Batch:
We want to process received stock quotes, once a minute, ordered by their stock symbol. We use XPath as the expression to select the stock symbol, as the value used for sorting.

```
from(“jms:topic:stock:quote”)
    .resequence().xpath(“/quote/@symbol”)
    .timeout(60 * 1000)
    .to(“seda:quotes”);
```

Camel will default the order to ascending. You can provide your own comparison for sorting if needed.

Stream:
Suppose we continuously poll a file directory for inventory updates, and it’s important they are processed in sequence by their inventory id. To do this we enable streaming and use one hour as the timeout.

```
from(“file://inventory”)
    .resequence().xpath(“/inventory/@id”)
    .stream().timeout(60 * 60 * 1000)
    .to(“seda:inventoryUpdates”);
```

XML DSL

Batch:

```
<route>
  <from uri=”jms:topic:stock:quote”/>
  <resequence>
    <xpath>/quote/@symbol</xpath>
    <batch-config batchTimeout=”60000”/>
  </resequence>
  <to uri=”seda:quotes”/>
</route>
```

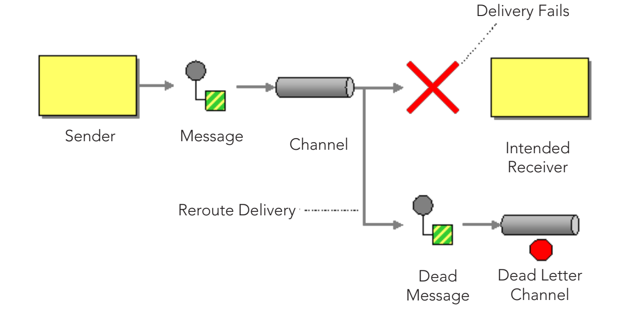
Stream:

```
<route>
  <from uri=”file://inventory”/>
  <resequence>
    <xpath>/inventory/@id</xpath>
    <stream-config timeout=”3600000”/>
  </resequence>
  <to uri=”seda:quotes”/>
</route>
```

Notice that you can enable streaming by specifying `<stream-config>` instead of `<batch-config>`.

Dead Letter Channel

What will the messaging system do with a message it cannot deliver?



Problem

The messaging system cannot deliver a message

Solution

When a message cannot be delivered it should be moved to a Dead Letter Channel

Camel

Camel has extensive support for Dead Letter Channel by its error handler and exception clauses. Error handler supports redelivery policies to decide how many times to try redelivering a message, before moving it to a Dead Letter Channel.

The default Dead Letter Channel will log the message at ERROR level and perform up to 6 redeliveries using a one second delay before each retry. Error handlers have two-level scope at either global or route.

TIP: The Camel in Action book, chapter 5 is devoted to cover error handling, which is the best source for information

Camel, continued

TIP: See Exception Clause for selective interception of thrown exception. This allows you to route certain exceptions differently or even reset the failure by marking it as handled.

TIP: DeadLetterChannel supports processing the message before it gets redelivered using onRedelivery. This allows you to alter the message beforehand (i.e. to set any custom headers).

Java DSL

Global scope
In global scope error handlers is defined before routes and applies to any routes which has not a route specific error handler.

```
errorHandler(deadLetterChannel(“file:error”)
  .maximumRedeliveries(3));

from(...)
```

Route scope
In route scope the error handler is defined inside the route and applies only to the given route.

```
from(“jms:queue:event”)
  .errorHandler(deadLetterChannel(“file:error/event”)
    .maximumRedeliveries(5).redeliveryDelay(5000))
  // and here begins the route
  .to(“log:event”)
  .to(“bean:handleEvent”);
```

XML DSL

Global scope
To use global scoped error handler you refer to it using the errorHandlerRef attribute on the <camelContext> tag as shown:

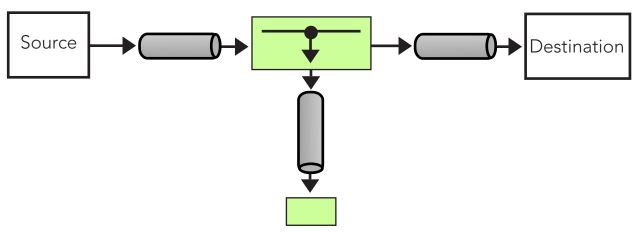
```
<camelContext errorHandlerRef=”eh”>
  <errorHandler id=”eh” type=”DeadLetterChannel”
deadLetterUri=”file:error”>
    <redeliveryPolicy maximumRedeliveries=”3”/>
  </errorHandler>
  <route>
     ...
  </route>
</camelContext>
```

Route scope
Route scope is likewise configured by referring to an error handler using errorHandlerRef attribute on the <route> tag as shown:

```
<route errorHandlerRef=”other-eh”>
  ...
</route>
```

Wire Tap

How do you inspect messages that travel on a point-to-point channel?



Problem

How do you tap messages while they are routed?

Solution

Insert a Wire Tap into the channel that publishes each incoming message to the main channel as well as to a secondary channel.

Camel

Camel has support for Wire Tap using the wireTap node that supports two modes: traditional and new message. The traditional mode sends a copy of the original message, as opposed to sending a new message. All messages are sent as Event Message and run in parallel with the original message.

Java DSL

Traditional
The route uses the traditional mode to send a copy of the original message to the seda tapped queue, while the original message is routed to its destination, the process order bean.

```
from(“jms:queue:order”)
    .wireTap(“seda:tappedOrder”)
    .to(“bean:processOrder”);
```

New message
In this route, we tap the high-priority orders and send a new message containing a body with the from part of the order.

Tip: As Camel uses an Expression for evaluation you can use other functions than **xpath**; for instance to send a fixed String, you can use **constant**.

```
from(“jms:queue:order”)
  .choice()
    .when(“/order/priority = ‘high’”)
       .wireTap(“seda:from”, xpath(“/order/from”))
       .to(“bean:processHighOrder”);
    .otherwise()
       .to(“bean:processOrder”);
```

XML DSL

Traditional

```
<route>
  <from uri="jms:queue:order"/>
  <wireTap uri="seda:tappedOrder"/>
  <to uri="bean:processOrder"/>
</route>
```

New Message

```
<route>
  <choice>
    <when>
      <xpath>/order/priority = 'high'</xpath>
      <wireTap uri="seda:from">
        <body><xpath>/order/from</xpath></body>
      </wireTap>
      <to uri="bean:processHighOrder"/>
    </when>
    <otherwise>
      <to uri="bean:processOrder"/>
    </otherwise>
  </choice>
</route>
```

CONCLUSION

The 12 patterns in this Refcard cover the often used patterns in the integration space. In this Refcard, you saw some of the great powers of the EIP patterns and what you can do when using them in practice with Apache Camel. You can find more examples of using EIPs at the Camel website: http://camel.apache.org/enterprise-integration-patterns.html. For more details about Camel, we highly recommend the book *Camel in Action*.

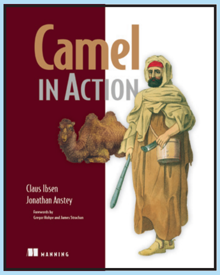
Get More Information

Camel website http://camel.apache.org	The home of the Apache Camel project. Find downloads, tutorials, examples, getting started guides, issue tracker, roadmap, and mailing lists.
FuseSource website http://fusesource.com	The home of the FuseSource company, the professional company behind Apache Camel with enterprise offerings, subscription, support, consulting, training, getting started guides, webinars, and tooling.
Camel in Action website http://manning.com/ibsen	The home of the Camel in Action book, published by Manning. The book is also on sale at Amazon and other retailers.
About Author http://davsclaus.blogspot.com	The personal blog of the author of this reference card. You can follow the author on twitter @davsclaus.

ABOUT THE AUTHOR

Claus Ibsen is a principal engineer working for FuseSource Corporation specializing in the enterprise integration space. Claus focuses mostly on Apache Camel and FUSE-related products. Claus has been engaged with Camel since late 2007, and he's co-author of the Camel in Action book, published by Manning. Claus is very active in the Apache and FUSE communities, writing blogs, twittering and assisting on the forums and irc channels. Claus is lead on Apache Camel and drives the roadmap. You will be able to meet Claus at various conferences where he speaks about Camel.

RECOMMENDED BOOK



Camel in Action is a Camel tutorial full of small examples showing how to work with the integration patterns. It starts with core concepts like sending, receiving, routing and transforming data. It then shows you the entire lifecycle and goes in depth on how to test, deal with errors, scale, deploy, and even monitor your app—details you can find only in the Camel code itself. Written by the developers of Camel, this book distills their experiences and practical insights so that you can tackle integration tasks like a pro.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-03-5
ISBN-10: 1-936502-03-8
50795
9 781936 502035
$7.95

Version 1.0