DZone Refcardz

Brought to you by...
Progress FUSE

Get More Refcardz! Visit refcardz.com

CONTENTS INCLUDE:

- About Enterprise Integration Patterns
- About Apache Camel
- Essential Patterns
- Conclusions and more...

Enterprise Integration Patterns

with Apache Camel

By Claus Ibsen

ABOUT ENTERPRISE INTEGRATION PATTERNS

Integration is a hard 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 Spring XML configuration. This reference card, the first in a two card series, guides you through the most common Enterprise Integration Patterns and gives you examples of how to implement them either in Java code or using Spring XML. This Refcard is targeted for software developers and enterprise architects, but anyone in the integration space can benefit as well.

ABOUT APACHE CAMEL

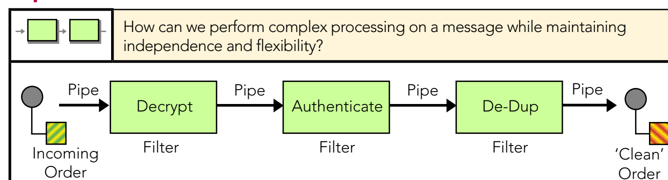
Apache Camel is a powerful open source integration platform based on Enterprise Integration Patterns (EIP) with powerful Bean Integration. Camel lets you implementing EIP routing using Camels intuitive Domain Specific Language (DSL) based on Java (aka fluent builder) or XML. Camel uses URI for endpoint resolution so its very easy to work with any kind of transport such as HTTP, REST, JMS, web service, File, FTP, TCP, Mail, JBI, Bean (POJO) and many others. Camel also provides Data Formats for various popular formats such as: CSV, EDI, FIX, HL7, JAXB, Json, Xstream. Camel is an integration API that can be embedded in any server of choice such as: J2EE Server, ActiveMQ, Tomcat, OSGi, or as standalone. Camels 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 plugin 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

How can we perform complex processing on a message while maintaining independence and flexibility?



Problem	A single event often triggers a sequence of processing steps
Solution	Use Pipes and Filters to divide a larger processing steps (filters) that are connected by channels (pipes)
Camel	Camel supports Pipes and Filters using the **pipeline** node.

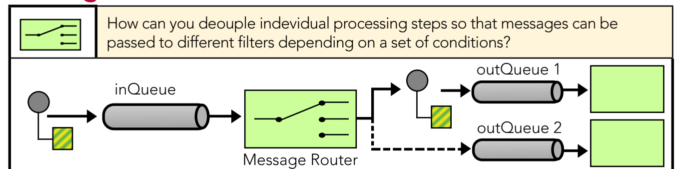
Java DSL	`from("jms:queue:order:in").pipeline("direct:transformOrder", "direct:validateOrder", "jms:queue:order:process");`
	Where jms represents the JMS component used for consuming JMS messages on the JMS broker. Direct is used for combining endpoints in a synchronous fashion, allow you to divide routes into sub routes and/or reuse common routes.
	Tip: Pipeline is the default mode of operation when you specify multiple outputs, so it can be omitted and replaced with the more common node:
	`from("jms:queue:order:in").to("direct:transformOrder", "direct:validateOrder", "jms:queue:order:process");`
	TIP: You can also separate each step as individual **to** nodes:
	`from("jms:queue:order:in")`
	`    .to("direct:transformOrder")`
	`    .to("direct:validateOrder")`
	`    .to("jms:queue:order:process");`

Spring DSL	See code below

```
<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>
```

Message Router

How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?



Problem	Pipes and Filters route each message in the same processing steps. How can we route messages differently?
Solution	Filter using predicates to choose the right output destination.
Camel	Camel supports Message Router using the **choice** node. For more details see the Content Based router pattern.

Are you using Apache Camel, Apache ActiveMQ, Apache ServiceMix, or Apache CXF?

Progress FUSE products are Apache-licensed, certified releases based on these Apache SOA projects.

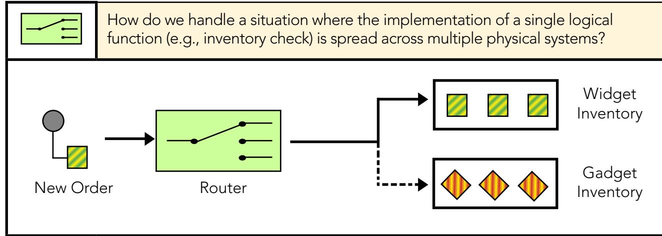
✓ Synchronized with Apache projects
✓ Enterprise support by Apache committers
✓ Online and on-site training

Register for a FREE webinar & learn how to build & deploy integration flows, web services & RESTful services with Apache Camel, Apache CXF, and Apache ServiceMix

Click here to register

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

Choice

```
from("jms:queue:order")
  .choice()
    .when(header("type").in("widget","wiggy"))
      .to("jms:queue:order:widget")
    .when(header("type").isEqualTo("gadget"))
      .to("jms:queue:order:gadget")
    .otherwise().to("jms:queue:order:misc")
  .end();
```

TIP: In the route above end() can be omitted as its the last node and we do not route the message to a new destination after the choice.

TIP: You can continue routing after the choice ends.

Spring DSL

Choice

```
<route>
  <from uri="jms:queue:order"/>
  <choice>
    <when>
      <simple>${header.type} in 'widget,wiggy'</simple>
      <to uri="jms:queue:order:widget"/>
    </when>
    <when>
      <simple>${header.type} == 'gadget'</simple>
      <to uri="jms:queue:order:gadget"/>
    </when>
    <otherwise>
      <to uri="jms:queue:order:misc"/>
    </otherwise>
  </choice>
</route>
```

TIP: In Spring DSL you cannot invoke code, as opposed to the Java DSL that is 100% Java. 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.

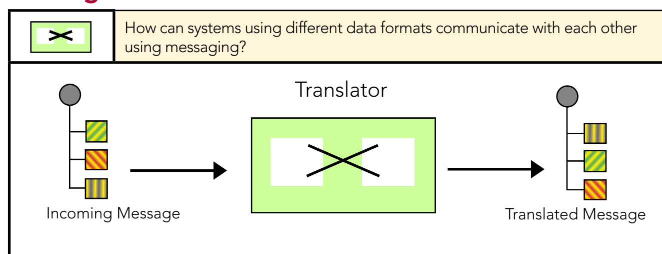
TIP: You can also use a method call to invoke a method on a bean to evaluate the predicate. Lets try that:

```
<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 messae 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

Processor

```
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"));
```

Spring DSL

Processor

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

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

In Spring 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>
```

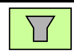
Annotation DSL

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.

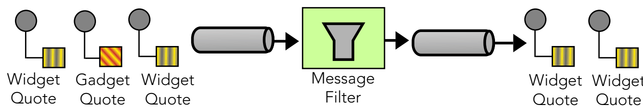
```
@Consume(uri="jms:queue:order:transform")
public String transformOrder(String body) {
    // do message translation
}
```

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** node. 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.

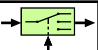
Java DSL
We want to discard any test messages so we only route non-test messages to the order queue.

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

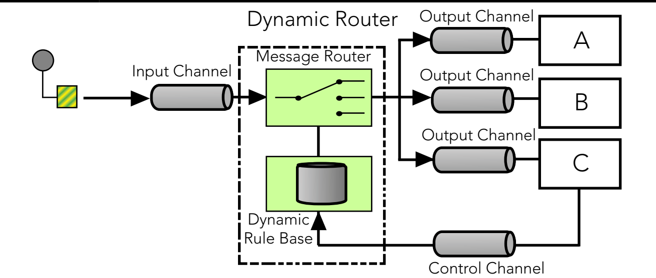
Spring DSL
For the Spring DSL we use XPath to evaluate the predicate. The $test is a special shorthand in Camel to refer to the header with the given name. So even if the payload is not XML based we can still use XPath to evaluate predicates.

```
<route>
  <from uri="jms:queue:inbox"/>
  <filter>
    <xpath>$test = 'false'</xpath>
    <to uri="jms:queue:inbox"/>
  </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 Dynamic **Recipient List** combined with a data store holding the list of destinations.

Java DSL
We use a Processor as the dynamic router to determine the destinations. We could also have used a Bean instead.

```
from("jms:queue:order")
    .processRef(myDynamicRouter)
    .recipientList("destinations");

public class MyDynamicRouter implements Processor {
    public void process(Exchange exchange) {
        // query a data store to find the best match of the
        // endpoint and return the destination(s) in the
        // header exchange.getIn()
        // .setHeader("destinations", list);
    }
}
```

Spring DSL

```
<route>
  <from uri="jms:queue:order"/>
  <process ref="myDynamicRouter"/>
  <recipientList>
    <header>destinations</destinations>
  </recipientList>
</route>
```

Annotation DSL

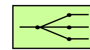
```
public class MyDynamicRouter {
    @Consume(uri = "jms:queue:order")
    @RecipientList
    public List<String> route(@XPath("/customer/id")
String customerId, @Header("location") String location,
Document body) {
        // query data store, find best match for the
        //endpoint and return destination (s)
    }
}
```

Annotation DSL, continued

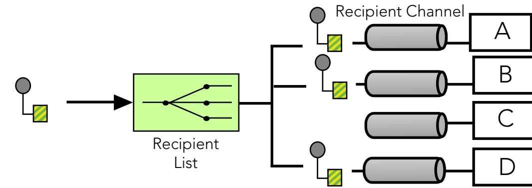
TIP: Notice how we used Bean Parameter Binding to bind the parameters to the route method based on an **@XPath** expression on the XML payload of the JMS message. This allows us to extract the customer id as a string parameter. **@Header** wil bind a JMS property with the key location. **Document** is the XML payload of the JMS message.

TIP: Camel uses its strong type converter feature to convert the payload to the type of the method parameter. We could use String and Camel will convert the body to a String instead. You can register your own type converters as well using the **@Converter** annotation at the class and method level.

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);
}
```

Spring 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", .... }
    }
}
```

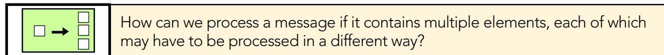
Annotation DSL
In the CustomerService class we annoate the `whereTo` method with **@RecipientList**, and return a single destination based on the customer id. Notice the flexibility of Camel as it can adapt accordingly to how you define what your methods are returning: a single element, a list, an iterator,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

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

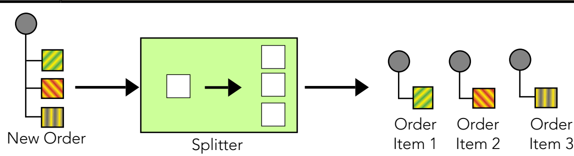
And then we can route to the bean and it will act as a dynamic recipient list.

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

Splitter



How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?



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");
```

Spring 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>
```

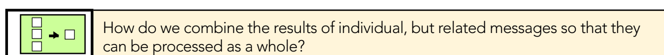
And in this route we split the messages using a regular expression

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

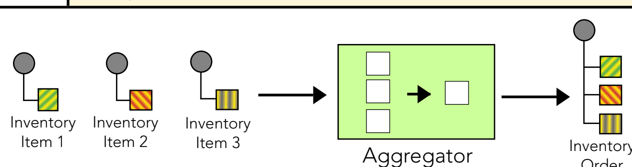
TIP: Split evaluates an `org.apache.camel.Expression` to provide something that is iterable to produce each individual new message. This allows you to provide any kind of expression such as a Bean invoked as a method call.

```
  <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** node. Camel uses a stateful batch processor that is capable of aggregating related messaged into a single combined message. A **correlation expression** is used to determine which messages should be aggregated. An **aggregation strategy** is used to combine aggregated messages into the result message. Camel's aggregator also supports a **completion predicate** allowing you to signal when the aggregation is complete. Camel also supports other completion signals based on timeout and/or a number of messages already aggregated.

Java DSL

Stock quote example

We want to update a website every five minutes 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")
    .batchTimeout(5 * 60 * 1000).to("seda: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. The time period is set as a timeout value in milliseconds.

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)
    .completionPredicate(header(Exchange.AGGREGATED_SIZE)
        .isGreaterThan(2))
    .to("seda:bestLoanQuote");
```

We use a **completion predicate** that signals when we have received more than 2 quotes for a given loan, giving us at least 3 quotes to pick 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;
    }
}
```

Spring DSL

Loan Broker Example

```
<route>
  <from uri="jms:topic:loan:qoute"/>
  <aggregate strategyRef="bestQuote">
    <correlationExpression>
      <header>loanId</header>
    </correlationExpression>
    <completionPredicate>
      <simple>${header.CamelAggregatedSize} > 2</simple>
    </completionPredicate>
  </aggregate>
  <to uri="seda:bestLoanQuote"/>
</route>

<bean id="bestQuote"
      class="com.mycompany.BestQuoteStrategy"/>
```

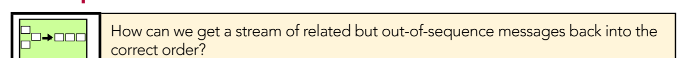
TIP: We use the **simple** language to declare the **completion predicate**. Simple is a basic language that supports a primitive set of operators. `${header.CamelAggregatedSize}` will fetch a header holding the number of messages aggregated.

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:

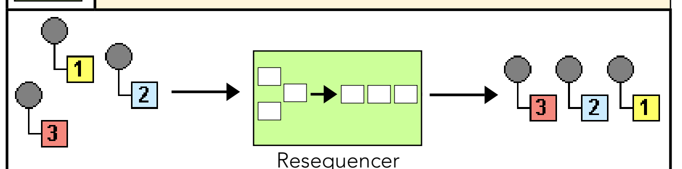
```
<completionPredicate>
  <method bean="quoteService" method="isComplete"/>
</compledtionPrediacate>

public boolean isComplete(@Header(Exchange.AGGREGATED_SIZE)
                          int count, String body) {
    return body.equals("STOP");
}
```

Notice how we can use Bean Binding Parameter to get hold of the aggregation size as a parameter, instead of looking it up in the message.

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

Camel, continued

supports two resequencing algorithms:

- **batch** = collects messages into a batch, sorts the messages and publish the messages
- **stream** = re-orders, 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 its 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");
```

Spring DSL

Batch:

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

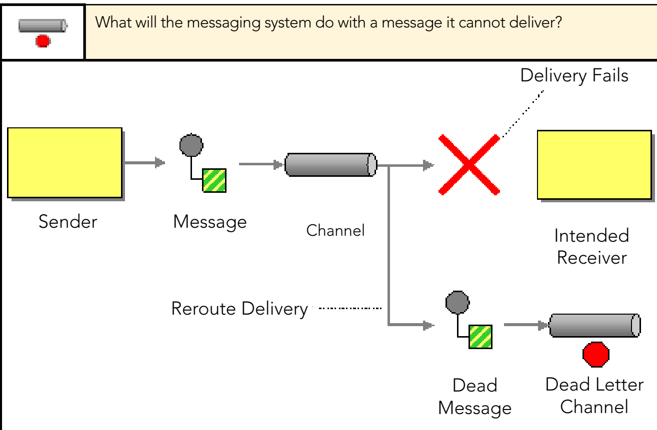
Stream:

```
<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 handler has two scopes: global and per route

TIP: See Exception Clause in the Camel documentation 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

```
errorHandler(deadLetterChannel("jms:queue:error")
    .maximumRedeliveries(3));

from(...)
```

Route scope

```
from("jms:queue:event")
    .errorHandler(deadLetterChannel()
    .maximumRedeliveries(5))
    .multicast().to("log:event", "seda:handleEvent");
```

In this route we override the global scope to use up to five redeliveries, where as the global only has three. You can of course also set a different error queue destination:

```
deadLetterChannel("log:badEvent").maximumRedeliveries(5)
```

Spring DSL

The error handler is configured very differently in the Java DSL vs. the Spring DSL. The Spring DSL relies more on standard Spring bean configuration whereas the Java DSL uses fluent builders.

Global scope
The Global scope error handler is configured using the **errorHandlerRef** attribute on the camelContext tag.

```
<camelContext errorHandlerRef="myDeadLetterChannel">
  ...
</camelContext>
```

Route scope
Route scoped is configured using the errorHandlerRef attribute on the route tag.

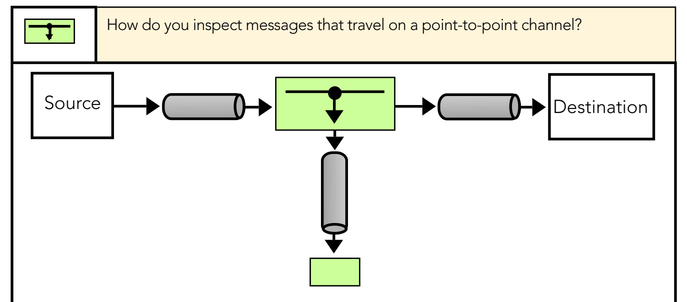
```
<route errorHandlerRef="myDeadLetterChannel">
  ...
</route>
```

For both the error handler itself is configured using a regular Spring bean

```
<bean id="myDeadLetterChannel" class="org.apache.camel.
builder.DeadLetterChannelBuilder">
  <property name="deadLetterUri" value="jms:queue:error"/>
  <property name="redeliveryPolicy"
       ref="myRedeliveryPolicy"/>
</bean>

<bean id="myRedeliverPolicy"
     class="org.apache.camel.processor.RedeliverPolicy">
  <property name="maximumRedeliveries" value="5"/>
  <property name="delay" value="5000"/>
</bean>
```

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 runs 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");
```

Spring 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>
```

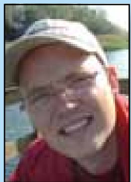
CONCLUSION

The twelve patterns in this Refcard cover the most used patterns in the integration space, together with two of the most complex such as the Aggregator and the Dead Letter Channel. In the second part of this series we will take a further look at common patterns and transations.

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, mailing lists, irc chat rooms, and how to get help.
FuseSource Website http://fusesource.com	The home of the FuseSource company, the professional company behind Apache Camel with enterprise offerings, support, consulting and training.
About Author http://davsclaus.blogspot.com	The personal blog of the author of this reference card.

ABOUT THE AUTHOR

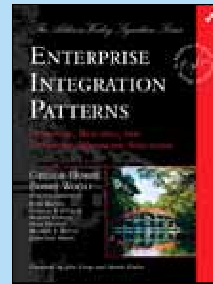


Claus Ibsen is a passionate open-source enthusiast who specializes in the integration space. As an engineer in the Progress FUSE open source team he works full time on Apache Camel, FUSE Mediation Router (based on Apache Camel) and related projects. Claus is very active in the Apache Camel and FUSE communities, writing blogs, twittering, assisting on the forums irc channels and is driving the Apache Camel roadmap.

ABOUT PROGRESS FUSE

FUSE products are standards-based, open source enterprise integration tools based on Apache SOA projects, and are productized and supported by the people who wrote the code.

RECOMMENDED BOOK

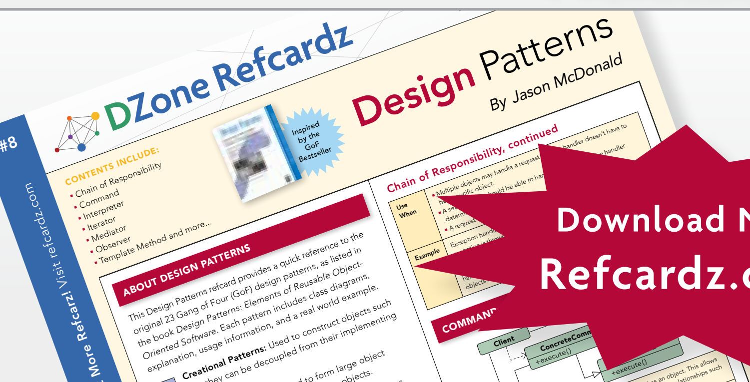


Utilizing years of practical experience, seasoned experts Gregor Hohpe and Bobby Woolf show how asynchronous messaging has proven to be the best strategy for enterprise integration success. However, building and deploying messaging solutions presents a number of problems for developers. Enterprise Integration Patterns provides an invaluable catalog of sixty-five patterns, with real-world solutions that demonstrate the formidable of messaging and help you to design effective messaging solutions for your enterprise.

BUY NOW

books.dzone.com/books/enterprise-integration-patterns

Professional Cheat Sheets You Can Trust



“Exactly what busy developers need: simple, short, and to the point.”

James Ward, Adobe Systems

Upcoming Titles

RichFaces
Agile Software Development
BIRT
JSF 2.0
Adobe AIR
BPM&BPMN
Flex 3 Components

Most Popular

Spring Configuration
jQuery Selectors
Windows Powershell
Dependency Injection with EJB 3
Netbeans IDE JavaEditor
Getting Started with Eclipse
Very First Steps in Flex

DZone communities deliver over 4 million pages each month to more than 2 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **“DZone is a developer’s dream,”** says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

Copyright © 2009 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher. Reference: Enterprise Integration Patterns, Gregor Hohpe, Addison-Wesley, 2003

Version 1.0