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. Hoppe & 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.

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 it's 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 plug your own component or data format, so they seamlessly blend in with existing modules. Camels provides a test kit for unit and integration testing with strong mock and assertion capabilities.

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

**Pipes and Filters**

- **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**

```java
```

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: 

```java
```

**Spring DSL**

```java
<route>
    <from uri="jms:queue:order:in"/>
    <pipeline>
        <to uri="direct:transformOrder"/>
        <to uri="jms:queue:order:process"/>
    </pipeline>
</route>
```

**Message Router**

How can you deouple 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.

**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.

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?

New Order | Router | Widget Inventory | Gadget Inventory

**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**

```java
from("jms:queue:order")
    .choice()
        .when(header("type").in("widget","wiggy"))
            .to("jms:queue:order:widget")
        .when(header("type").isEqual("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**

```java
@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>
```

**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. Let's try that:

```java
@method bean="myBean" method="isGadget"/>
```

**TIP:** Spring DSL 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:

```java
@method transformer="OrderTransformerBean.transformOrder"/>
```

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

**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.

**Java DSL**

```java
@Consume
    public boolean isGadget(@Header(name = "type") String type) {
        return type.equals("Gadget");
    }
```

**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.`

**Spring DSL**

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

**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:

```java
@method bean="OrderTransformerBean.class"/>
```

**TIP:** In Spring DSL you can 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.

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

**TIP:** Camel supports the message translator using the `producer, bean or transform` nodes.

**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.

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

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

**Problem**
How can we route messages based on a static or 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 supports the Dynamic Recipient List combined with a data store holding the list of destinations. We can route to the bean and it will act as a dynamic recipient list.

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

```java
public class MyDynamicRouter implements Processor {
    // 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**

```java
<route>
   <from uri="jms:queue:inbox"/>
   <recipientList>
    <header>destinations</header>
   </recipientList>
</route>
```

**Annotation DSL**

```java
public class MyDynamicRouter {
    @Consume(uri = "jms:queue:order")
    @RecipientList
    public String route(@Path("customerId") String customerId, @Header("location") String location) {
        // query data store, find best match for the
        // endpoint and return destination(s)
    }
}
```

**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 method on a bean to provide the dynamic list of destinations.

**Java DSL**
Static

```java
@method bean = "myDynamicRouter" method="route"/>
</recipientList>
</route>
```

**Spring DSL**

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

**Annotation DSL**

```java
@method bean = "myDynamicRouter" method="route"/>
</recipientList>
</route>
```

In the CustomerService class we annotate the whereTo method with @RecipientList and return a single destination based on the customerId.

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.

```java
public class CustomerService {
    @RecipientList
    public String whereTo(@Header("customerId") id) {
        return id;
    }
}
```

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

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

**Message Filter**

How can a component avoid receiving unwanted messages?

**Problem**
How can a component avoid receiving 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 supports the 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.

```java
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.

```
<from uri="jms:queue:inbox"/>
   <filter>
      <xpath>$test = 'false'</xpath>
   </filter>
</route>
```

**Dynamic Router**

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

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

**Solution**
Use a Processor as the dynamic router to determine the destinations. We could also have used a Bean instead.

```
processRef(myDynamicRouter)
   .recipientList("destinations");
```

**Java DSL Static**

```java
processMails = ()
 .multiCast().to("file://backup", "seda:inbox");
```

**Spring DSL Static**

```java
<route>
   <from uri="jms:queue:inbox"/>
   <recipientList>
    <header>destinations</header>
   </recipientList>
</route>
```

**Annotation DSL**

In the CustomerService class we annotate the whereTo method with @RecipientList and return a single destination based on the customerId.

```
@method bean = "myDynamicRouter" method="route"/>
</recipientList>
</route>
```

```java
from("jms:queue:inbox")
   .beanRef(processMails)
   .recipientList("destinations");
```

Notice how we used Bean Parameter Binding to bind the parameters to the route method based on an @Path expression on the XML payload of the JMS message. This allows us to extract the customer id as a string parameter. @Header will 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.
**Aggregator**

How can 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 messages 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.

**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 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 redelivers using a one second delay before each retry. Error handler has two scopes: global and per route.

**Java DSL**
```
errorHandler(deadLetterChannel("jms:queue:error")
    .maximumRedeliveries(3));
from(...)
```

**Spring DSL**
```
The error handler is configured very differently in the Java DSL vs. the Spring DSL. The Spring DSL relies on standard Spring bean configuration whereas the Java DSL uses fluent builders.
```
```java
globalScope = CamelContext.errorHandlerRef("myDeadLetterChannel")
```
```
routeScope = CamelContext.errorHandlerRef(attribute on the route tag)
```
```
from("jms:queue:order")
    .errorHandlerRef("myDeadLetterChannel")
    .maximumRedeliveries(5)
    .multicast().to("log:event", "seda:handleEvent");
```

In this route we override the global scope to use up to five redeliveries, whereas the global only has three. You can of course also set a different error queue destination:
```
deadLetterChannel("log:badEvent").maximumRedeliveries(5)
```

### Dead Letter Channel

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

**Source** | **Destination**
---|---
**Message** | **Dead Letter Channel**
**Channel** | **Intended Receiver**
**Reroute Delivery** | **Dead Message**

**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 redelivers 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 route the failure by marking it as handled.

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

**Java DSL**
```
errorHandler(deadLetterChannel("jms:queue:error")
    .maximumRedeliveries(5));
from(...)
```

**Spring DSL**
```
<route scope="global">
    <from uri="jms:queue:order"/>
    <errorHandlerRef="seda:tappedOrder"/>
</route>
```

In this route we override the global scope to use up to five redeliveries, whereas the global only has three. You can of course also set a different error queue destination:
```
deadLetterChannel("log:badEvent").maximumRedeliveries(5)
```

### Wire Tap

**How do you tap messages while they are routed?**

**Source** | **Destination**
---|---
**Message** | **Dead Letter Channel**
**Channel** | **Intended Receiver**
**Reroute Delivery** | **Dead Message**

**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**
```
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. Tap: 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.
```
<route scope="stream">
    <from uri="jms:queue:event">
        <errorHandlerRef="myDeadLetterChannel">...
    </from>
</route>
```

**Spring DSL**
```
wireTap = CamelContext.wireTapRef("seda:tappedOrder")
```
```
wireTap = CamelContext.wireTapRef("seda:tappedOrder")
```
```
<route scope="stream">
    <from uri="jms:queue:event">
        <errorHandlerRef="myDeadLetterChannel">...
    </from>
</route>
```

**Problem**

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

**Solution**

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

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://davisclaus.blogspot.com

The personal blog of the author of this reference card.

ABOUT DESIGN PATTERNS

The original 23 Gang of Four (GoF) design patterns, as listed in the book Oriented Software by 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&8PMN
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

Download Now
Reocardz.com

DZone Refcardz

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.