User Guide

Version 1.0-SNAPSHOT

Table of Contents

Chapter 1. Introduction

Apache Camel is a powerful rule based routing and mediation engine which provides a POJO based implementation of the Enterprise Integration Patterns using an extremely powerful fluent API (or declarative Java Domain Specific Language) to configure routing and mediation rules. The Domain Specific Language means that Apache Camel can support type-safe smart completion of routing rules in your IDE using regular Java code without huge amounts of XML configuration files; though Xml Configuration inside Spring is also supported.

Apache Camel uses generics, annotations and URIs so that it can easily work directly with any kind of Transport or messaging model such as HTTP, JMS, JBI, SCA, MINA or CXF Bus API without mandating a normalized message API which can often lead to leaky abstractions. Apache Camel is also a small library which has minimal dependencies for easy embedding in any Java application.

Apache Camel can be used as a routing and mediation engine for the following projects:

- Apache ActiveMQ which is the most popular and powerful open source message broker
- Apache CXF which is a smart web services suite (JAX-WS)
- Apache MINA a networking framework
- Apache ServiceMix which is the most popular and powerful distributed open source ESB and JBI container

So don't get the hump, try Camel today!

Chapter 2. Architecture

Routing Domain Specific Language (DSL)Xml Configurationrouting and mediation rulesCamelContextEnterprise Integration Patterns

At a high level Camel consists of a CamelContext which contains a collection of Component instances. A Component is essentially a factory of Endpoint instances. You can explicitly configure Component instances in Java code or an IoC container like Spring or Guice, or they can be auto-discovered using URIs.

An Endpoint acts rather like a URI or URL in a web application or a Destination in a JMS system; you can communicate with an endpoint; either sending messages to it or consuming messages from it. You can then create a Producer or Consumer on an Endpoint to exchange messages with it.

The DSL makes heavy use of pluggable Languages to create an Expression or Predicate to make a truly powerful DSL which is extensible to the most suitable language depending on your needs. The following languages are supported

- Scripting Languages such as
  - BeanShell
  - JavaScript
  - Groovy
  - Python
  - PHP
  - Ruby
- SQL
- XPath
- XQuery

Camel makes extensive use of URIs to allow you to refer to endpoints which are lazily created by a Component if you refer to them within Routes

Component	URI	Description
ActiveMQ	`activemq:[topic:]destinationName`	For JMS Messaging with Apache ActiveMQ
CXF	`cxf:serviceName`	Working with Apache CXF for web services integration
Direct	`direct:name`	Direct invocation of the consumer from the producer so that single threaded (non-SEDA) in VM invocation is performed

File	`file://nameOfFileOrDirectory`	Sending messages to a file or polling a file or directory
FTP	`ftp://host[:port]/fileName`	Sending and receiving files over FTP
HTTP	`http://hostname[:port]`	Working with the HTTP protocol either consuming requests over HTTP or consuming external RESTful resources
IMap	`imap://hostname[:port]`	Receiving email using IMap
IRC	`irc:host[:port]/#room`	For IRC communication
JBI	`jbi:serviceName`	For JBI integration such as working with Apache ServiceMix
JMS	`jms:[topic:]destinationName`	Working with JMS providers
JPA	`jpa://entityName`	For using a database as a queue via the JPA specification for working with OpenJPA, Hibernate or TopLink
Mail	`mail://user-info@host:port`	Sending and receiving email
MINA	`[tcp|udp|multicast]:host[:port]`	Working with Apache MINA
Multicast	`multicast://host:port`	Working with TCP protocols using Apache MINA
Mock	`mock:name`	For testing routes and mediation rules using mocks
Pojo	`pojo:name`	Exposing and invoking a POJO
POP	`pop3://user-info@host:port`	Receiving email using POP3 and JavaMail
Quartz	`quartz://groupName/timerName`	Provides a scheduled delivery of messages using the Quartz scheduler
Queue	`queue:name`	Used to deliver messages to a java.util.Queue, useful when creating SEDA style processing

		pipelines
RMI	`rmi://host[:port]`	Working with RMI
SFTP	`sftp://host[:port]/fileName`	Sending and receiving files over SFTP
SMTP	`smtp://user-info@host[:port]`	Sending email using SMTP and JavaMail
Timer	`timer://name`	A timer endpoint
TCP	`tcp://host:port`	Working with TCP protocols using Apache MINA
UDP	`udp://host:port`	Working with UDP protocols using Apache MINA
XMPP	`xmpp://host:port/room`	Working with XMPP and Jabber
WebDAV	`webdav://host[:port]/fileName`	Sending and receiving files over WebDAV

For a full details of the individual components see the Component Appendix

Chapter 3. Enterprise Integration Patterns

Camel supports most of the Enterprise Integration Patterns from the excellent book of the same name by Gregor Hohpe and Bobby Woolf. Its a highly recommended book, particularly for users of Camel.

There now follows a list of the Enterprise Integration Patterns from the book along with examples of the various patterns using Apache Camel

	Message Channel	How does one application communicate with another using messaging?
	Message	How can two applications connected by a message channel exchange a piece of information?
	Pipes and Filters	How can we perform complex processing on a message while maintaining independence and flexibility?
	Message Router	How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?
	Message Translator	How can systems using different data formats communicate with each other using messaging?
	Message Endpoint	How does an application connect to a messaging channel to send and receive messages?

Messaging Channels

	Point to Point Channel	How can the caller be sure that exactly one receiver will receive the document or perform the call?
	Publish Subscribe Channel	How can the sender broadcast an event to all interested receivers?
	Dead Letter Channel	What will the messaging system do with a message it

		cannot deliver?
	Guaranteed Delivery	How can the sender make sure that a message will be delivered, even if the messaging system fails?
	Message Bus	What is an architecture that enables separate applications to work together, but in a de-coupled fashion such that applications can be easily added or removed without affecting the others?

Message Routing

	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?
	Message Filter	How can a component avoid receiving uninteresting messages?
	Recipient List	How do we route a message to a list of dynamically specified recipients?
	Splitter	How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?
	Resequencer	How can we get a stream of related but out-of-sequence messages back into the correct order?

Message Transformation

	Content Enricher	How do we communicate with another system if the message originator does not have all the required data items available?

	Content Filter	How do you simplify dealing with a large message, when you are interested only in a few data items?
	Normalizer	How do you process messages that are semantically equivalent, but arrive in a different format?

Messaging Endpoints

	Messaging Mapper	How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?
	Event Driven Consumer	How can an application automatically consume messages as they become available?
	Polling Consumer	How can an application consume a message when the application is ready?
	Competing Consumers	How can a messaging client process multiple messages concurrently?
	Message Dispatcher	How can multiple consumers on a single channel coordinate their message processing?
	Selective Consumer	How can a message consumer select which messages it wishes to receive?
	Durable Subscriber	How can a subscriber avoid missing messages while it's not listening for them?
	Idempotent Consumer	How can a message receiver deal with duplicate messages?
	Transactional Client	How can a client control its transactions with the messaging system?
	Messaging Gateway	How do you encapsulate access to the messaging system from the rest of the

		application?
	Service Activator	How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?

System Management

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

For a full breakdown of each pattern see the Book Pattern Appendix

Chapter 4. Getting Started with Apache Camel

Chapter 5. Pattern Appendix

There now follows a breakdown of the various Enterprise Integration Patterns that Camel supports

Camel supports the Message Channel from the EIP patterns. The Message Channel is an internal implementation detail of the Endpoint interface and all interactions with the Message Channel are via the Endpoint interfaces.

For more details see

- Message
- Message Endpoint

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Message

Camel supports the Message from the EIP patterns using the Message interface.

To support various message exchange patterns like one way event messages and request-response messages Camel uses an Exchange interface which is used to handle either oneway messages with a single inbound Message, or request-reply where there is an inbound and outbound message.

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Pipes and Filters

Camel supports the Pipes and Filters from the EIP patterns in various ways.

With Camel you can split your processing across multiple independent Endpoint instances which can then be chained together.

Using Routing Logic

You can create pipelines of logic using multiple Endpoint or Message Translator instances as follows

```
from("direct:a").pipeline("direct:x", "direct:y", "direct:z", "mock:result");
```

In the above example we are routing from a single Endpoint to a list of different endpoints specified using URIs. If you find the above a bit confusing, try reading about the Architecture or try the

Examples

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Message Router

The Message Router from the EIP patterns allows you to consume from an input destination, evaluate some predicate then choose the right output destination.

The following example shows how to route a request from an input **queue:a** endpoint to either **queue:b**, **queue:c** or **queue:d** depending on the evaluation of various Predicate expressions

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").choice()
                .when(header("foo").isEqualTo("bar")).to("queue:b")
                .when(header("foo").isEqualTo("cheese")).to("queue:c")
                .otherwise().to("queue:d");
    }
};
```

Using the Spring XML Extensions

```
<camelContext id="buildSimpleRouteWithChoice" xmlns="http://activemq.apache.org/camel/schema/spring"><route><fro
```

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Message Translator

Camel supports the Message Translator from the EIP patterns by using an artibrary Processor in the routing logic

Using the Fluent Builders

```
from("direct:start").setBody(body().append(" World!")).to("mock:result");
```

or you can add your own Processor

```
from("direct:start").process(new Processor() {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}).to("mock:result");
```

For further examples of this pattern in use you could look at one of the JUnit tests

- TransformTest
- TransformViaDSLTest

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Message Endpoint

Camel supports the Message Endpoint from the EIP patterns using the Endpoint interface.

When using the DSL to create Routes you typically refer to Message Endpoints by their URIs rather than directly using the Endpoint interface. Its then a responsibility of the CamelContext to create and activate the necessary Endpoint instances using the available Component implementations.

For more details see

- Message

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Messaging Channels

Point to Point Channel

Camel supports the Point to Point Channel from the EIP patterns using the following components

- Queue for in-VM seda based messaging
- JMS for working with JMS Queues for high performance, clustering and load balancing
- JPA for using a database as a simple message queue
- XMPP for point-to-point communication over XMPP (Jabber)

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of

the Examples first before trying this pattern out.

Publish Subscribe Channel

Camel supports the Publish Subscribe Channel from the EIP patterns using the following components

- JMS for working with JMS Topics for high performance, clustering and load balancing
- XMPP when using rooms for group communication

Using Routing Logic

Another option is to explicitly list the publish-subscribe relationship in your routing logic; this keeps the producer and consumer decoupled but lets you control the fine grained routing configuration using the DSL or Xml Configuration.

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").to("queue:b", "queue:c", "queue:d");
    }
};
```

Using the Spring XML Extensions

```
<camelContext id="buildStaticRecipientList" xmlns="http://activemq.apache.org/camel/schema/spring"><route><from
```

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Dead Letter Channel

Camel supports the Dead Letter Channel from the EIP patterns using the DeadLetterChannel processor which is an Error Handler.

Redelivery

It is common for a temporary outage or database deadlock to cause a message to fail to process; but the chances are if its tried a few more times with some time delay then it will complete fine. So we typically wish to use some kind of redelivery policy to decide how many times to try redeliver a message and how long to wait before redelivery attempts.

The RedeliveryPolicy defines how the message is to be redelivered. You can customize things like

- how many times a message is attempted to be redelivered before it is considered a failure and

sent to the dead letter channel

- the initial redelivery timeout
- whether or not exponential backoff is used (i.e. the time between retries increases using a backoff multiplier)
- whether to use collision avoidence to add some randomness to the timings

Once all attempts at redelivering the message fails then the message is forwarded to the dead letter queue.

Redelivery header

When a message is redelivered the DeadLetterChannel will append a customizable header to the message to indicate how many times its been redelivered. The default value is **org.apache.camel.redeliveryCount**.

Configuring via the DSL

The following example shows how to configure the Dead Letter Channel configuration using the DSL

```
RouteBuilder<Exchange> builder = new RouteBuilder<Exchange>() {
    public void configure() {
        errorHandler(deadLetterChannel("queue:errors"));

        from("queue:a").to("queue:b");
    }
};
```

You can also configure the RedeliveryPolicy as this example shows

```
RouteBuilder<Exchange> builder = new RouteBuilder<Exchange>() {
    public void configure() {
        errorHandler(deadLetterChannel("queue:errors").maximumRedeliveries(2).useExponentialBackOff());

        from("queue:a").to("queue:b");
    }
};
```

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Guaranteed Delivery

Camel supports the Guaranteed Delivery from the EIP patterns using the following components

- File for using file systems as a persistent store of messages
- JMS when using persistent delivery (the default) for working with JMS Queues and Topics for high performance, clustering and load balancing

- JPA for using a database as a persistence layer

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Message Bus

Camel supports the Message Bus from the EIP patterns. You could view Camel as a Message Bus itself as it allows producers and consumers to be decoupled.

Folks often assume that a Message Bus is a JMS though so you may wish to refer to the JMS component for traditional MOM support.

Also worthy of node is the XMPP component for supporting messaging over XMPP (Jabber)

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Message Routing

Content Based Router

The Content Based Router from the EIP patterns allows you to route messages to the correct destination based on the contents of the message exchanges.

The following example shows how to route a request from an input **queue:a** endpoint to either **queue:b**, **queue:c** or **queue:d** depending on the evaluation of various Predicate expressions

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").choice()
                .when(header("foo").isEqualTo("bar")).to("queue:b")
                .when(header("foo").isEqualTo("cheese")).to("queue:c")
                .otherwise().to("queue:d");
    }
};
```

Using the Spring XML Extensions

```
<camelContext id="buildSimpleRouteWithChoice" xmlns="http://activemq.apache.org/camel/schema/spring"><route><fro
```

For further examples of this pattern in use you could look at the junit test case

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Message Filter

The Message Filter from the EIP patterns allows you to filter messages

The following example shows how to create a Message Filter route consuming messages from an endpoint called **queue:a** which if the Predicate is true will be dispatched to **queue:b**

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").filter(header("foo").isEqualTo("bar")).to("queue:b");
    }
};
```

You can of course use many different Predicate languages such as XPath, XQuery, SQL or various Scripting Languages. Here is an XPath example

```
from("direct:start").filter(
        xpath("/person[@name='James']")
).to("mock:result");
```

Using the Spring XML Extensions

```
<camelContext id="buildSimpleRouteWithHeaderPredicate" xmlns="http://activemq.apache.org/camel/schema/spring"><r
```

For further examples of this pattern in use you could look at the junit test case

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Recipient List

The Recipient List from the EIP patterns allows you to route messages to a number of destinations.

Static Receipient List

The following example shows how to route a request from an input **queue:a** endpoint to a static list of destinations

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
```

```
        from("queue:a").to("queue:b", "queue:c", "queue:d");
    }
};
```

Using the Spring XML Extensions

```
<camelContext id="buildStaticRecipientList" xmlns="http://activemq.apache.org/camel/schema/spring"><route><from
```

Dynamic Recipient List

Usually one of the main resons for using the Recipient List pattern is that the list of recipients is dynamic and calculated at runtime. The following example demostrates how to create a dynamic recipient list using an Expression (which in this case it extracts a named header value dynamically) to calculate the list of endpoints which are either of type Endpoint or are converted to a String and then resolved using the endpoint URIs.

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").recipientList(header("foo"));
    }
};
```

The above assumes that the header contains a list of endpoint URIs. The following takes a single string header and tokenizes it

```
from("direct:a").recipientList(header("recipientListHeader").tokenize(","));
```

Using the Spring XML Extensions

```
<camelContext id="buildDynamicRecipientList" xmlns="http://activemq.apache.org/camel/schema/spring"><route><from
```

For further examples of this pattern in use you could look at one of the junit test case

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Splitter

The Splitter from the EIP patterns allows you split a message into a number of pieces and process them individually

Example

The following example shows how to take a request from the **queue:a** endpoint the split it into pieces using an Expression, then forward each piece to **queue:b**

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").splitter(bodyAs(String.class).tokenize("\n")).to("queue:b");
    }
};
```

The splitter can use any Expression language so you could use any of the Languages Supported such as XPath, XQuery, SQL or one of the Scripting Languages to perform the split. e.g.

```
from("activemq:my.queue").splitter(xpath("//foo/bar")).to("file://some/directory")
```

Using the Spring XML Extensions

```
<camelContext id="buildSplitter" xmlns="http://activemq.apache.org/camel/schema/spring"><route><from uri="queue:
                <tokenize token="
"/>
            </recipients></splitter><to uri="queue:b"/></route></camelContext>
```

For further examples of this pattern in use you could look at one of the junit test case

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Resequencer

The Resequencer from the EIP patterns allows you to reorganise messages based on some comparator. By default in Camel we use an Expression to create the comparator; so that you can compare by a message header or the body or a piece of a message etc.

The following example shows how to reorder the messages so that they are sorted in order of the **body()** expression. That is messages are collected into a batch (either by a maximum number of messages per batch or using a timeout) then they are sorted in order and then sent out to their output.

Using the Fluent Builders

```
from("direct:a").resequencer(body()).to("mock:result");
```

So the above example will reorder messages from endpoint **direct:a** in order of their bodies, to the endpoint **mock:result**. Typically you'd use a header rather than the body to order things; or maybe a part of the body. So you could replace this expression with

```
resequencer(header("JMSPriority"))
```

for example to reorder messages using their JMS priority.

You can of course use many different Expression languages such as XPath, XQuery, SQL or various Scripting Languages.

You can also use multiple expressions; so you could for example sort by priority first then some other custom header

```
resequencer(header("JMSPriority"), header("MyCustomerRating"))
```

Using the Spring XML Extensions

For further examples of this pattern in use you could look at the junit test case

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Message Transformation

Content Enricher

Camel supports the Content Enricher from the EIP patterns using a Message Translator or by using an arbitrary Processor in the routing logic to enrich the message.

Using the Fluent Builders

Here is a simple example using the DSL directly

```
from("direct:start").setBody(body().append(" World!")).to("mock:result");
```

In this example we add our own Processor

```
from("direct:start").process(new Processor() {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}).to("mock:result");
```

For further examples of this pattern in use you could look at one of the JUnit tests

- TransformTest
- TransformViaDSLTest

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Content Filter

Camel supports the Content Filter from the EIP patterns using a Message Translator or by using an artibrary Processor in the routing logic to filter content from the inbound message.

A common way to filter messages is to use an Expression in the DSL like XQuery, SQL or one of the supported Scripting Languages.

Using the Fluent Builders

Here is a simple example using the DSL directly

```
from("direct:start").setBody(body().append(" World!")).to("mock:result");
```

In this example we add our own Processor

```
from("direct:start").process(new Processor() {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}).to("mock:result");
```

For further examples of this pattern in use you could look at one of the JUnit tests

- TransformTest
- TransformViaDSLTest

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Normalizer

Camel supports the Normalizer from the EIP patterns by using a Message Router in front of a number of Message Translator instances.

See Also

- Message Router
- Content Based Router
- Message Translator

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Messaging Endpoints

Messaging Mapper

Camel supports the Messaging Mapper from the EIP patterns by using either Message Translator pattern or the Type Converter module.

See also

- Message Translator
- Type Converter
- CXF for JAX-WS support for binding business logic to messaging & web services
- POJO
- Bean

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Event Driven Consumer

Camel supports the Event Driven Consumer from the EIP patterns. The default consumer model is event based (i.e. asynchronous) as this means that the Camel container can then manage pooling, threading and concurrency for you in a declarative manner.

The Event Driven Consumer is implemented by consumers implementing the Processor interface which is invoked by the Message Endpoint when a Message is available for processing.

For more details see

- Message
- Message Endpoint

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Polling Consumer

Camel supports implementing the Polling Consumer from the EIP patterns using the

PollingConsumer interface which can be created via the Endpoint.createPollingConsumer() method.

So in your Java code you can do

```
Endpoint endpoint = context.getEndpoint("activemq:my.queue");
PollingConsumer consumer = endpoint.createPollingConsumer();
Exchange exchange = consumer.receive();
```

There are 3 main polling methods on PollingConsumer

Method name	Description
receive()	Waits until a message is available and then returns it; potentially blocking forever
receive(long)	Attempts to receive a message exchange immediately without waiting and returning null if a message exchange is not available yet
receiveNoWait()	Attempts to receive a message exchange, waiting up to the given timeout and returning null if no message exchange could be received within the time available

Scheduled Poll Components

Quite a few inbound Camel endpoints use a scheduled poll pattern to receive messages and push them through the Camel processing routes. That is to say externally from the client the endpoint appears to use an Event Driven Consumer but internally a scheduled poll is used to monitor some kind of state or resource and then fire message exchanges.

Since this a such a common pattern, polling components can extend the ScheduledPollConsumer base class which makes it simpler to implement this pattern.

There is also the Quartz Component which provides scheduled delivery of messages using the Quartz enterprise scheduler.

For more details see

- PollingConsumer
- Scheduled Polling Components
  - ScheduledPollConsumer
  - File
  - JPA
  - Mail
  - Quartz

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Competing Consumers

Camel supports the Competing Consumers from the EIP patterns using a few different components.

You can use the following components to implement competing consumers:-

- Queue for SEDA based concurrent processing using a thread pool
- JMS for distributed SEDA based concurrent processing with queues which support reliable load balancing,ß failover and clustering.

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Message Dispatcher

Camel supports the Message Dispatcher from the EIP patterns using various approaches.

You can use a component like JMS with selectors to implement a Selective Consumer as the Message Dispatcher implementation. Or you can use an Endpoint as the Message Dispatcher itself and then use a Content Based Router as the Message Dispatcher.

See Also

- JMS
- Selective Consumer
- Content Based Router
- Endpoint

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Selective Consumer

The Selective Consumer from the EIP patterns can be implemented in two ways

The first solution is to provide a Message Selector to the underlying URIs when creating your consumer. For example when using JMS you can specify a selector parameter so that the message broker will only deliver messages matching your criteria.

The other approach is to use a Message Filter which is applied; then if the filter matches the message your consumer is invoked as shown in the following example

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").filter(header("foo").isEqualTo("bar")).process(myProcessor);
    }
};
```

Using the Spring XML Extensions

```
<camelContext id="buildCustomProcessorWithFilter" xmlns="http://activemq.apache.org/camel/schema/spring"><route>
```

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Durable Subscriber

Camel supports the Durable Subscriber from the EIP patterns using the JMS component which supports publish & subscribe using Topics with support for non-durable and durable subscribers.

Another alternative is to combine the Message Dispatcher or Content Based Router with File or JPA components for durable subscribers then something like Queue for non-durable.

See Also

- JMS
- File
- JPA
- Message Dispatcher
- Selective Consumer
- Content Based Router
- Endpoint

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Idempotent Consumer

The Idempotent Consumer from the EIP patterns is used to filter out duplicate messages.

This pattern is implemented using the IdempotentConsumer class. This uses an Expression to calculate a unique message ID string for a given message exchange; this ID can then be looked up in the MessageIdRepository to see if it has been seen before; if it has the message is consumed; if its not then the message is processed and the ID is added to the repository.

The Idempotent Consumer essentially acts like a Message Filter to filter out duplicates.

Using the Fluent Builders

The following example will use the header **myMessageId** to filter out duplicates

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").idempotentConsumer(
                header("myMessageId"), memoryMessageIdRepository(200)
        ).to("queue:b");
    }
};
```

The above ??? will use an in-memory based MessageIdRepository which can easily run out of memory and doesn't work in a clustered environment. So you might prefer to use the JPA based implementation which uses a database to store the message IDs which have been processed

```
returnnew SpringRouteBuilder() {
    public void configure() {
        from("direct:start").idempotentConsumer(
                header("messageId"),
                jpaMessageIdRepository(bean(JpaTemplate.class), "myProcessorName")
        ).to("mock:result");
    }
};
```

In the above ??? we are using the header **messageId** to filter out duplicates and using the collection **myProcessorName** to indicate the Message ID Repository to use. This name is important as you could process the same message by many different processors; so each may require its own logical Message ID Repository.

Using the Spring XML Extensions

```
<camelContext id="buildCustomProcessorWithFilter" xmlns="http://activemq.apache.org/camel/schema/spring"><route>
```

For further examples of this pattern in use you could look at the junit test case

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of

the Examples first before trying this pattern out.

Transactional Client

Camel recommends supporting the Transactional Client from the EIP patterns using spring transactions.

Transaction Oriented Endpoints (Camel Toes) like JMS support using a transaction for both inbound and outbound message exchanges. Endpoints that support transactions will participate in the current transaction context that they are called from.

You should use the SpringRouteBuilder to setup the routes since you will need to setup the spring context with the TransactionTemplates that will define the transaction manager configuration and policies.

For inbound endpoint to be transacted, they normally need to be configured to use a Spring PlatformTransactionManager. In the case of the JMS component, this can be done by looking it up in the spring context.

You first define needed object in the spring configuration.

```
<bean id="jmsTransactionManager" class="org.springframework.jms.connection.JmsTransactionManager"><property name
```

Then you look them up and use them to create the JmsComponent.

```
PlatformTransactionManager transactionManager = (PlatformTransactionManager) spring.getBean("jmsTransactionManag
  ConnectionFactory connectionFactory = (ConnectionFactory) spring.getBean("jmsConnectionFactory");
  JmsComponent component = JmsComponent.jmsComponentTransacted(connectionFactory, transactionManager);
  component.getConfiguration().setConcurrentConsumers(1);
  ctx.addComponent("activemq", component);
```

Transaction Policies

Outbound endpoints will automatically enlist in the current transaction context. But what if you do not want your outbound endpoint to enlist in the same transaction as your inbound endpoint? The solution is to add a Transaction Policy to the processing route. You first have to define transaction policies that you will be using. The policies use a spring TransactionTemplate to declare the transaction demarcation use. So you will need to add something like the following to your spring xml:

```
<bean id="PROPAGATION_REQUIRED" class="org.springframework.transaction.support.TransactionTemplate"><property na
```

Then in your SpringRouteBuilder, you just need to create new SpringTransactionPolicy objects for each of the templates.

```
public void configure() {
   ...
   Policy requried = new SpringTransactionPolicy(bean(TransactionTemplate.class, "PROPAGATION_REQUIRED"));
   Policy notsupported = new SpringTransactionPolicy(bean(TransactionTemplate.class, "PROPAGATION_NOT_SUPPORTED"
   Policy requirenew = new SpringTransactionPolicy(bean(TransactionTemplate.class, "PROPAGATION_REQUIRES_NEW"));
   ...
}
```

Once created, you can use the Policy objects in your processing routes:

```
// Send to bar in a new transaction
   from("activemq:queue:foo").policy(requirenew).to("activemq:queue:bar");
```

```
// Send to bar without a transaction.
from("activemq:queue:foo").policy(notsupported ).to("activemq:queue:bar");
```

See Also

- JMS

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Messaging Gateway

Camel has several endpoint components that support the Messaging Gateway from the EIP patterns.

Components like Bean, CXF and Pojo provide a a way to bind a Java interface to the message exchange.

See Also

- Bean
- Pojo
- CXF

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Service Activator

Camel has several endpoint components that support the Service Activator from the EIP patterns.

Components like Bean, CXF and Pojo provide a a way to bind the message exchange to a Java interface/service.

See Also

- Bean
- Pojo

- CXF

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

System Management

Wire Tap

The Wire Tap from the EIP patterns allows you to route messages to a separate tap location before it is forwarded to the ultimate destination.

The following example shows how to route a request from an input **queue:a** endpoint to the wire tap location **queue:tap** before it is received by **queue:b**

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").to("queue:tap", "queue:b");
    }
};
```

Using the Spring XML Extensions

```
<camelContext id="buildWireTap" xmlns="http://activemq.apache.org/camel/schema/spring"><route><from uri="queue:a
```

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

Chapter 6. Camel Components

There now follows the documentation on each Camel component.

The ActiveMQ component allows messages to be sent to a JMS Queue or Topic ; or messages to be consumed from a JMS Queue or Topic using Apache ActiveMQ. This component is based on the JMS Component and uses Spring's JMS support for declarative transactions, using Spring's JmsTemplate for sending and a MessageListenerContainer for consuming.

```
activemq:[topic:]destinationName
```

So for example to send to queue FOO.BAR you would use

```
activemq:FOO.BAR
```

You can be completely specific if you wish via

```
activemq:queue:FOO.BAR
```

If you want to send to a topic called Stocks.Prices then you would use

```
activemq:topic:Stocks.Prices
```

Configuring the Connection Factory

The following ??? shows how to add an ActiveMQComponent to the CamelContext using the activeMQComponent() method while specifying the brokerURL used to connect to ActiveMQ

```
camelContext.addComponent("activemq", activeMQComponent("vm://localhost?broker.persistent=false"));
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

CXF Component

The **cxf:** component provides integration with Apache CXF for connecting to JAX-WS services hosted in CXF.

URI format

```
cxf:address
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

Direct Component

The **direct:** component provides direct, synchronous invocation of any consumers when a producer sends a message exchange.This endpoint can be used connect existing routes or if a client in the same JVM as the Camel router wants to access the routes.

URI format

```
direct:someName
```

Where **someName** can be any string to uniquely identify the endpoint

Options

Name	Default Value	Description
allowMultipleConsumers	true	If set to false, then when a second consumer is started on the endpoint, a IllegalStateException is thrown

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

File Component

The *??? component provides access to file systems.

URI format

```
file:fileName
```

Where **fileName** represents the underlying file name

Options

Name	Default Value	Description
initialDelay	1000	milliseconds before polling the file/directory starts
delay	500	milliseconds before the next poll of the file/directory
useFixedDelay	false	if true, poll once after the initial delay
recursive	true	if a directory, will look for changes in files in all the sub directories
attemptFileLock	false	if true, will only fire an exchange for a file it can lock
regexPattern	null	will only fire a an exchange for a file that matches the regex pattern

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

FTP/SFTP/WebDAV Component

This component provides access to remote file systems over the FTP, SFTP and WebDAV protocols

URI format

```
ftp://host[:port]/fileName[?options]
sftp://host[:port]/fileName[?options]
webdav://host[:port]/fileName[?options]
```

Where **fileName** represents the underlying file name or directory

Options

Name	Default Value	Description
directory	false	indicates whether or not the given file name should be interpreted by default as a directory or file (as it sometimes hard to be sure with some FTP servers)
password	null	specifies the password to use to login to the remote file system

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

HTTP Component

The **http:** component provides HTTP based endpoints for exposing HTTP resources or consuming external HTTP resources.

URI format

```
http:hostname[:port][/resourceUri]
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JBI Component

The **jbi:** component provides integration with a JBI Service Bus such as provided by Apache ServiceMix

URI format

```
jbi:service:serviceQName
jbi:interface:interfaceQName
jbi:endpoint:endpointName
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JMS Component

The JMS component allows messages to be sent to a JMS Queue or Topic; or messages to be consumed from a JMS Queue or Topic. The implementation of the JMS Component uses Spring's JMS support for declarative transactions, using Spring's JmsTemplate for sending and a MessageListenerContainer for consuming.

URI format

```
jms:[topic:]destinationName?properties
```

So for example to send to queue FOO.BAR you would use

```
jms:FOO.BAR
```

You can be completely specific if you wish via

```
jms:queue:FOO.BAR
```

If you want to send to a topic called Stocks.Prices then you would use

```
jms:topic:Stocks.Prices
```

Notes

If you wish to use durable topic subscriptions, you need to specify both **clientId** and **durableSubscriberName**. Note that the value of the clientId must be unique and can only be used by a single JMS connection instance in your entire network. You may prefer to use Virtual Topics instead to avoid this limitation. More background on durable messaging here.

	If you are using ActiveMQ Note that the JMS component reuses Spring 2's

	JmsTemplate for sending messages. This is not ideal for use in a non-J2EE container and typically requires some caching JMS provider to avoid performance being lousy. So if you intent to use Apache ActiveMQ as your Message Broker - which is a good choice as ActiveMQ rocks , then we recommend that you either • use the ActiveMQ component which is already configured to use ActiveMQ efficiently • use the PoolingConnectionFactory in ActiveMQ

Properties

You can configure lots of different properties on the JMS endpoint which map to properties on the JMSConfiguration POJO.

Property	Default Value	Description
acceptMessagesWhileStopping	false	Should the consumer accept messages while it is stopping
acknowledgementModeName	"AUTO_ACKNOWLEDGE"	The JMS acknowledgement name which is one of: TRANSACTED, CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE
autoStartup	true	Should the consumer container auto-startup
cacheLevelName	"CACHE_CONSUMER"	Sets the cache level name for the underlying JMS resources
clientId	null	Sets the JMS client ID to use. Note that this value if specified must be unique and can only be used by a single JMS connection instance. Its typically only required for durable topic subscriptions. You may prefer to use Virtual Topics instead
concurrentConsumers	1	Specifies the default number of

		concurrent consumers
connectionFactory	null	The default JMS connection factory to use for the listenerConnectionFactory and templateConnectionFactory if neither are specified
deliveryPersistent	true	Is persistent delivery used by default?
durableSubscriptionName	null	The durable subscriber name for specifying durable topic subscriptions
exceptionListener	null	The JMS Exception Listener used to be notified of any underlying JMS exceptions
explicitQosEnabled	false	Set if the deliveryMode, priority or timeToLive should be used when sending messages
exposeListenerSession	true	Set if the listener session should be exposed when consuming messages
idleTaskExecutionLimit	1	Specify the limit for idle executions of a receive task, not having received any message within its execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in case of dynamic scheduling; see the "maxConcurrentConsumers" setting).
listenerConnectionFactory	null	The JMS connection factory used for consuming messages
maxConcurrentConsumers	1	Specifies the maximum number of concurrent consumers
maxMessagesPerTask	1	The number of messages per task
messageConverter	null	The Spring Message Converter
messageIdEnabled	true	When sending, should message IDs be added
messageTimestampEnabled	true	Should timestamps be enabled by default on sending messages

priority	-1	Values of > 1 specify the message priority when sending, if the explicitQosEnabled property is specified
receiveTimeout	none	The timeout when receiving messages
recoveryInterval	none	The recovery interval
serverSessionFactory	null	The JMS ServerSessionFactory if you wish to use ServerSessionFactory for consumption
subscriptionDurable	false	Enabled by default if you specify a durableSubscriberName and a clientId
taskExecutor	null	Allows you to specify a custom task executor for consuming messages
templateConnectionFactory	null	The JMS connection factory used for sending messages
timeToLive	null	Is a time to live specified when sending messages
transacted	false	Is transacted mode used for sending/receiving messages?
transactionManager	null	The Spring transaction manager to use
transactionName	null	The name of the transaction to use
transactionTimeout	null	The timeout value of the transaction if using transacted mode
useVersion102	false	Should the old JMS API be used

See Also

- Configuring Camel
- Component
- Endpoint

- Getting Started

JPA Component

The **jpa:** component allows you to work with databases using JPA (EJB 3 Persistence) such as for working with OpenJPA, Hibernate, TopLink to work with relational databases.

Sending POJOs to the JPA endpoint inserts entities into the database. Consuming messages removes (or updates) entities in the database.

URI format

```
jpa:entityClassName
```

Options

Name	Default Value	Description

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

Mail Component

The **mail:** component provides access to Email via Spring's Mail support and the underlying JavaMail system

URI format

```
pop://[user-info@]host[:port][?password=somepwd]
imap://[user-info@]host[:port][?password=somepwd]
smtp://[user-info@]host[:port][?password=somepwd]
```

which supports either POP, IMAP or SMTP underlying protocols.

Property	Description
host	the host name or IP address to connect to
port	the TCP port number to connect on
user-info	the user name on the email server

password	the users password to use

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

MINA Component

The **mina:** component is a transport for working with Apache MINA

URI format

```
mina:tcp://hostname[:port]
mina:udp://hostname[:port]
mina:multicast://hostname[:port]
```

Options

Name	Default Value	Description

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

Mock Component

The **mock:** component provides a powerful declarative testing mechanism which is similar to jMock in that it allows declarative expectations to be created on an endpoint up front, then a route used, then the expectations can be asserted in a test case to ensure the routing rules and processors worked as expected.

URI format

```
mock:someName
```

Where **someName** can be any string to uniquely identify the endpoint

Examples

Here's quick example of MockEndpoint in use, asserting the number of messages which are expected during a test run

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class);

resultEndpoint.expectedMessageCount(2);

// send some messages
...

// now lets assert that the mock:foo endpoint received 2 messages
resultEndpoint.assertIsSatisfied();
```

You can see from the javadoc of MockEndpoint the various helper methods you can use. You can use other methods such as

- assertIsSatisfied()
- expectedMessageCount(int)
- expectedBodiesReceived(...)

Here's another example

```
resultEndpoint.expectedBodiesReceived("firstMessageBody", "secondMessageBody", "thirdMessageBody");
```

Or you could add an expectation on the headers or content of a specific message. For example

```
resultEndpoint.message(0).header("foo").isEqualTo("bar");
```

There are some examples of the Mock endpoint in use in the camel-corecore processor tests.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

Pojo Component

The **pojo:** component binds PojoExchanges to method invocations on Java Objects.

URI format

```
pojo:someName
```

Where **someName** can be any string to uniquely identify the endpoint

Using

Object instance that can receive invocations, must be explicitly registered with the PojoComponent.

```
PojoComponent component = (PojoComponent)camelContext.getComponent("pojo");
component.addService("bye", new SayService("Good Bye!"));
```

Once an endpoint has been registered, you can build Camel routes that use it to process exchanges.

```
// lets add simple route
camelContext.addRoutes(new RouteBuilder() {
    public void configure() {
        from("direct:hello").to("pojo:bye");
    }
});
```

A **pojo:** endpoint cannot be defined as the input to the route. Consider using a **direct:** or **queue:** endpoint as the input for a PojoExchange. You can use the createProxy() methods on PojoComponent to create a proxy that will generate PojoExchanges and send them to any endpoint:

```
Endpoint endpoint = camelContext.getEndpoint("direct:hello");
ISay proxy = PojoComponent.createProxy(endpoint, ISay.class);
String rc = proxy.say();
assertEquals("Good Bye!", rc);
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

Quartz Component

The **quartz:** component provides a scheduled delivery of messages using the Quartz scheduler.Each endpoint represents a different timer (in Quartz terms, a Trigger and JobDetail).

URI format

```
quartz://timerName?parameters
quartz://groupName/timerName?parameters
quartz://groupName/timerName/cronExpression
```

You can configure the Trigger and JobDetail using the parameters

Property	Description
trigger.repeatCount	How many times should the timer repeat for?
trigger.repeatInterval	The amount of time in milliseconds between repeated triggers
job.name	Sets the name of the job

For example the following routing rule will fire 2 timer events to the endpoint **mock:results**

```
from("quartz://myGroup/myTimerName?trigger.repeatInterval=2&trigger.repeatCount=1").to("mock:result");
```

Using Cron Triggers

Quartz supports Cron-like expressions for specifying timers in a handy format. You can use these expressions in the URI; though to preserve valid URI encoding we allow / to be used instead of spaces and $ to be used instead of ?.

For example the following will fire a message at 12pm (noon) every day

```
from("quartz://myGroup/myTimerName/0/0/12/*/*/$").to("activemq:Totally.Rocks");
```

which is equivalent to using the cron expression

```
0 0 12 * * ?
```

The following table shows the URI character encodings we use to preserve valid URI syntax

URI Character	Cron character
'/'	' '
'$'	'?'

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

Queue Component

The **queue:** component provides asynchronous SEDA behaviour so that consumers are invoked in a seperate thread pool to the producer within the same JVM. Note this component has nothing to do

with JMS, if you want a distributed SEA then try using either JMS or ActiveMQ or even MINA

URI format

```
queue:someName
```

Where **someName** can be any string to uniquely identify the endpoint

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

RMI Component

The **rmi:** component bind the PojoExchanges to the RMI protocol (JRMP).

Since this binding is just using RMI, normal RMI rules still apply in regards to what the methods can be used over it. This component only supports PojoExchanges that carry a method invocation that is part of an interface that extends the Remote interface. All parameters in the method should be either Serializable or Remote objects too.

URI format

```
rmi://rmi-regisitry-host:rmi-registry-port/registry-path
```

For example:

```
rmi://localhost:1099/path/to/service
```

Using

To call out to an existing RMI service registered in an RMI registry, create a Route similar to:

```
from("pojo:foo").to("rmi://localhost:1099/foo");
```

To bind an existing camel processor or service in an RMI registry, create a Route like:

```
RmiEndpoint endpoint= (RmiEndpoint) endpoint("rmi://localhost:1099/bar");
endpoint.setRemoteInterfaces(ISay.class);
from(endpoint).to("pojo:bar");
```

Notice that when binding an inbound RMI endpoint, the Remote interfaces exposed must be specified.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

Timer Component

The **timer:** component derives from the POJO component to provide timed events. You can only consume events from this endpoint. It produces POJO exchanges that send a Runnable.run() method invocation.

URI format

```
timer:name?options
```

Where **options** is a query string that can specify any of the following parameters:

Name	Default Value	Description
time		The date/time that the (first) event should be generated.
period	-1	If set to greater than 0, then generate periodic events every period milliseconds
delay	-1	The number of milliseconds to wait before the first event is generated. Should not be used in conjunction with the time parameter.
fixedRate	false	Events take place at approximately regular intervals, separated by the specified period.
daemon	true	Should the thread associated with the timer endpoint be run as a daemon.

Using

To setup a route that generates an event every 500 seconds:

```
from("timer://foo?fixedRate=true&delay=0&period=500").to("pojo:bar");
```

Note that the "bar" pojo registered should implement Runnable.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

XMPP Component

The **xmpp:** component implements an XMPP (Jabber) transport.

URI format

```
xmpp:hostname[:port][/room]
```

The component supports both room based and private person-person conversations

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started