Fuse Mediation Router

Component Reference

Version 2.6
January 2011

Component Reference

Version 2.6

Updated: 03 Dec 2013

Trademark Disclaimer

Third Party Acknowledgements

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (http://jline.sourceforge.net) jline:jline:jar:1.0

  License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux `<mwp1@cornell.edu>`

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (http://woodstox.codehaus.org/StAX2) org.codehaus.woodstox:stax2-api:jar:3.1.1

  License: The BSD License (http://www.opensource.org/licenses/bsd-license.php)

  Copyright (c) <YEAR>, <OWNER> All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (http://www.jibx.org/main-reactor/jibx-run) org.jibx:jibx-run:bundle:1.2.3

  License: BSD (http://jibx.sourceforge.net/jibx-license.html) Copyright (c) 2003-2010, Dennis M. Sosnoski.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (http://www.jboss.org/javassist) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile

  License: MPL (http://www.mozilla.org/MPL/MPL-1.1.html)

- HAPI-OSGI-Base Module (http://hl7api.sourceforge.net/hapi-osgi-base/) ca.uhn.hapi:hapi-osgi-base:bundle:1.2

  License: Mozilla Public License 1.1 (http://www.mozilla.org/MPL/MPL-1.1.txt)

Table of Contents

List of Tables

List of Examples

Chapter 1. Components Overview

This chapter provides a summary of all the components available for Fuse Mediation Router.

List of Components

Table of components

The following components are available for use with Fuse Mediation Router.

Table 1.1. Fuse Mediation Router Components

Component	Endpoint URI	Artifact ID	Description
ActiveMQ	activemq:[queue:|topic:]*destinationName*	`activemq-core`	For JMS Messaging with Apache ActiveMQ.
ActiveMQ Journal	activemq.journal:*directoryName*[?*options*]	`activemq-core`	Uses ActiveMQ's fast disk journaling implementation to store message bodies in a rolling log file.
AMQP	amqp:[queue:|topic:]*destinationName*[?*options*]]	`camel-amqp`	For messaging with the AMQP protocol.
Atom	`atom://`*AtomUri*[?*Options*]	`camel-atom`	Working with Apache Abdera for atom integration, such as consuming an atom feed.
Bean	bean:*BeanID*[?methodName=*Method*]	`camel-core`	Uses the Bean Binding to bind message exchanges to beans in the Registry. Is also used for exposing and invoking POJO (Plain Old Java Objects).
Bean Validation	`bean-validator:`*Some thing*[?*Options*]	`camel-bean-validator`	Validates the payload of a message using the Java Validation API (JSR 303[1] and JAXP Validation) and its reference implementation Hibernate Validator[2].
Browse	`browse:`*Name*	`camel-core`	Provdes a simple `BrowsableEndpoint` which can be useful for testing,

[1] http://jcp.org/en/jsr/detail?id=303
[2] http://docs.jboss.org/hibernate/stable/validator/reference/en/html_single/

Component	Endpoint URI	Artifact ID	Description
			visualisation tools or debugging. The exchanges sent to the endpoint are all available to be browsed.
Cache	cache://*CacheName*[?*Options*]	camel-cache	The cache component enables you to perform caching operations using EHCache as the Cache Implementation.
Cometd	cometd://*Hostname*[:*Port*]/*ChannelName*[?*Options*]	camel-cometd	A transport for working with the jetty implementation of the cometd/bayeux protocol.
Crypto	crypto:sign:*Name*[?*Options*] crypto:verify:*Name*[?*Options*]	camel-crypto	Sign and verify exchanges using the Signature Service of the Java Cryptographic Extension.
CXF	cxf://*Address*[?*Options*]	camel-cxf	Working with Apache CXF for web services integration.
CXFRS	cxfrs:bean:*RsEndpoint*[?*Options*]	camel-cxf	Provides integration with Apache CXF for connecting to JAX-RS services hosted in CXF.
DataSet	dataset:*Name*[?*Options*]	camel-core	For load & soak testing the DataSet provides a way to create huge numbers of messages for sending to Components or asserting that they are consumed correctly.
Direct	direct:*EndpointID*[?*Options*]	camel-core	Direct invocation of the consumer from the producer so that single threaded (non-SEDA) in VM invocation is performed.

Component	Endpoint URI	Artifact ID	Description
Esper	`esper:name`	`camel-esper`	Working with the Esper Library for Event Stream Processing.
Event	`event://dummy`	`camel-spring`	Working with Spring ApplicationEvents.
EventAdmin on page 121	`eventadmin:topic`	`camel-eventadmin`	
Exec	`exec://Executable[?Options]`	`camel-exec`	Execute system command.
File2	`file://DirectoryName[?Options]`	`camel-core`	Sending messages to a file or polling a file or directory.
FIX	`fix://ConfigurationResource`	`camel-fix`	Sends or receives messages using the FIX protocol.
Flatpack	`flatpack:[fixed|delim]:ConfigFile`	`camel-flatpack`	Processing fixed width or delimited files or messages using the FlatPack[3] library
Freemarker	`freemarker:TemplateResource`	`camel-freemarker`	Generates a response using a Freemarker[4] template.
FTP2	`ftp://[username@]hostname[:port]/directoryname[?options]`	`camel-ftp`	Sending and receiving files over FTP.
GAuth	`gauth://Name[?Options]`	`camel-gae`	Used by web applications to implement a Google-specific OAuth[5] consumer

GHTTP	`ghttp:///Path[?Options]`
	`ghttp://Host name[:Port]/Path[?Op tions]`
	`ghttps://Host name[:Port]/Path[?Op tions]`
		`camel-gae`	Provides connectivity to the GAE URL fetch service and can also be used to receive messages from servlets.

GLogin	`glogin://Hostname[:Port][?Options]`	`camel-gae`	Used by Camel applications outside Google App Engine (GAE)

[3] http://flatpack.sourceforge.net/
[4] http://freemarker.org/
[5] http://code.google.com/apis/accounts/docs/OAuth.html

Component	Endpoint URI	Artifact ID	Description
			for programmatic login to GAE applications.
GMail	`gmail://User name@gmail.com[?Options]` `gmail://Username@googlemail.com[?Options]`	`camel-gae`	Supports sending of emails via the GAE mail service.
GTask	`gtask://QueueName`	`camel-gae`	Supports asynchronous message processing on GAE using the task queueing service as a message queue.
HDFS	`hdfs://Path[?Options]`	`camel-hdfs`	For reading/writing from/to an HDFS[6] filesystem.
Hibernate	`hibernate://EntityName`	`camel-hibernate` (Camel Extra)	For using a database as a queue via the Hibernate[7] library.
HL7	`mina:tcp://Host[:Port]`	`camel-hl7`	For working with the HL7 MLLP protocol and the HL7 model using the HAPI library[8].
HTTP	`http://Hostname[:Port][/ResourceUri]`	`camel-http`	For calling out to external HTTP servers.
iBATIS	`ibatis:OperationName[?Options]`	`camel-ibatis`	Performs a query, poll, insert, update or delete in a relational database using Apache iBATIS.
IMap	`imap://[UserName@]Host[:Port][?Options]`	`camel-mail`	Receiving email using IMap.
IRC	`irc:Host[:Port]/#Room`	`camel-irc`	For IRC communication.
JavaSpace	`javaspace:jini://Host[?Options]`	`camel-javaspace`	Sending and receiving messages through JavaSpace.

[6] http://http/hadoop.apache.org/hdfs/
[7] http://www.hibernate.org/
[8] http://hl7api.sourceforge.net/

Component	Endpoint URI	Artifact ID	Description

JBI	`jbi:service:serviceNamespace[sep]serviceName`
	`jbi:endpoint:serviceNamespace[sep]serviceName[sep]endpointName`
	`jbi:name:endpointName`
		`camel-jbi`	For JBI integration such as working with Apache ServiceMix.

JCR	`jcr://userName:password@repository/pathToNode`	`camel-jcr`	Storing a message in a JCR (JSR-170) compliant repository like Apache Jackrabbit.
JDBC	`jdbc:dataSourceName[?options]`	`camel-jdbc`	For performing JDBC queries and operations.
JDBC Aggregator Repository on page 271		`camel-jdbc-aggregator`	
Jetty	`jetty:http://host[:port][/resourceUri]`	`camel-jetty`	For exposing services over HTTP.
Jing	`rng:LocalOrRemoteResource` `rnc:LocalOrRemoteResource`	`camel-jing`	Validates the payload of a message using RelaxNG or RelaxNG compact syntax.
JMS	`jms:[temp:][queue:|topic:]destinationName[?options]`	`camel-jms`	Working with JMS providers.
JPA	`jpa:[EntityClassName][?options]`	`camel-jpa`	For using a database as a queue via the JPA specification for working with OpenJPA, Hibernate or TopLink.
JT400	`jt400://user:pwd@system/pathToDTAQ`	`camel-jt400`	For integrating with data queues on an AS/400 (aka System i, IBM i, i5, ...) system.
LDAP	`ldap:host[:port]?base=...[&scope=Scope]`	`camel-ldap`	Performing searches on LDAP servers (*Scope* must be one of `object|onelevel|subtree`).
List	`list:ListID`	`camel-core`	Provides a simple BrowsableEndpoint which can be useful for testing,

Component	Endpoint URI	Artifact ID	Description
			visualisation tools or debugging. The exchanges sent to the endpoint are all available to be browsed.
Log	log:*LoggingCategory*[?level=*LoggingLevel*]	camel-core	Uses Jakarta Commons Logging to log the message exchange to some underlying logging system like log4j.
Lucene	lucene:*SearcherName*:insert[?analyzer=*Analyzer*] lucene:*SearcherName*:query[?analyzer=*Analyzer*]	camel-lucene	Uses Apache Lucene to perform Java-based indexing and full text based searches using advanced analysis/tokenization capabilities.
MINA	mina:tcp://*HostName*[:*Port*][?*Options*] mina:udp://*HostName*[:*Port*][?*Options*] mina:multicast://*HostName*[:*Port*][?*Options*] mina:vm://*HostName*[:*Port*][?*Options*]	camel-mina	Working with Apache MINA.
Mock	mock:*EndpointID*	camel-core	For testing routes and mediation rules using mocks.
MSMQ	msmq:*MsmqQueueName*[?*Options*]	camel-msmq	Sending and receiving messages with Microsoft Message Queuing.
MSV	msv:*LocalOrRemoteResource*	camel-msv	Validates the payload of a message using the MSV Library.
Nagios	nagios://*Host*[:*Port*][?*Options*]	camel-nagios	Sending passive checks to Nagios using JSendNSCA[9].

[9] http://code.google.com/p/jsendnsca/

Component	Endpoint URI	Artifact ID	Description
Netty	`netty:tcp://localhost:99999[?Options]` `netty:udp://Remote host:99999/[?Options]`	`camel-netty`	Working with TCP and UDP protocols using Java NIO based capabilities offered by the JBoss Netty[10] community project.
NMR	`nmr:serviceMixURI`	`servicemix-camel`	For OSGi integration when working with Apache ServiceMix. Enables you to specify the URI of a ServiceMix endpoint[11].
Pax-Logging on page 379	`paxlogging:Appender`	`camel-paxlogging`	
POP	`pop3://[UserName@]Host[:Port][?Options]`	`camel-mail`	Receiving email using POP3 and JavaMail.
Printer	`lpr://localhost[:Port]/default[?Options]` `lpr://RemoteHost[:Port]/path/to/printer[?Options]`	`camel-printer`	Provides a way to direct payloads on a route to a printer.
Properties	`properties://Key[?Options]`	`camel-properties`	Facilitates using property placeholders directly in endpoint URI definitions.
Quartz	`quartz://[GroupName/]TimerName[?Options]` `quartz://GroupName/TimerName/CronExpression`	`camel-quartz`	Provides a scheduled delivery of messages using the Quartz scheduler.
Quickfix	`quickfix-server:ConfigFile` `quickfix-client:ConfigFile`	`camel-quickfix`	Implementation of the QuickFix for Java engine which allow to send/receive FIX[12] messages.
Ref	`ref:EndpointID`	`camel-core`	Component for lookup of existing endpoints bound in the Registry.

[10] http://www.jboss.org/netty
[11] http://servicemix.apache.org/uris.html
[12] http://www.fixprotocol.org/

Component	Endpoint URI	Artifact ID	Description
Restlet	`restlet:RestletUrl[?Options]`	`camel-restlet`	Component for consuming and producing Restful resources using Restlet[13].
RMI	`rmi://[RegistryHost[:RegistryPort]/]RegistryPath`	`camel-rmi`	Working with RMI.
Routebox on page 415	`routebox:routeboxName[?Options]`	`camel-routebox`	
RSS	`rss:Uri`	`camel-rss`	Working with ROME[14] for RSS integration, such as consuming an RSS feed.
RNC	`rnc:LocalOrRemoteResource`	`camel-jing`	Validates the payload of a message using RelaxNG Compact Syntax.
RNG	`rng:LocalOrRemoteResource`	`camel-jing`	Validates the payload of a message using RelaxNG.
Scalate	`scalate:TemplateName[?Options]`	`org.fusesource.scalate/scalate-camel`	Uses the given Scalate[15] template to transform the message.
SEDA	`seda:EndpointID`	`camel-core`	Used to deliver messages to a `java.util.concurrent.BlockingQueue`, useful when creating SEDA style processing pipelines within the same CamelContext.
SERVLET	`servlet://RelativePath[?Options]`	`camel-servlet`	Provides HTTP based endpoints for consuming HTTP requests that arrive at a HTTP endpoint and this endpoint is bound to a published Servlet.
SFTP	`sftp://[username@]host[:port]/directoryname[?Options]`	`camel-ftp`	Sending and receiving files over SFTP.
Smooks			For working with EDI parsing using the Smooks library[16].

[13] http://www.restlet.org/
[14] https://rome.dev.java.net/
[15] http://scalate.fusesource.org/
[16] http://milyn.codehaus.org/Smooks

Component	Endpoint URI	Artifact ID	Description
SMPP	smpp://*UserInfo*@*Host*[:*Port*][?*Options*]	camel-smpp	To send and receive SMS using Short Messaging Service Center using the JSMPP library[17].
SMTP	smtp://[*UserName*@]*Host*[:*Port*][?*Options*]	camel-mail	Sending email using SMTP and JavaMail.
SNMP	snmp://*Hostname*[:*Port*][?*Options*]	camel-snmp	Gives you the ability to poll SNMP capable devices or receive traps.
Spring Integration	spring-integration:*DefaultChannelName*[?*Options*]	camel-spring-integration	The bridge component of Camel and Spring Integration.
SQL	sql:*SqlQueryString*[?*Options*]	camel-sql	Performing SQL queries using JDBC.
Stream	stream:[in|out|err|header][?*Options*]	camel-stream	Read or write to an input/output/error/file stream rather like Unix pipes.
String Template	string-template:*TemplateURI*[?*Options*]	camel-stringtemplate	Generates a response using a String Template.
Test	test:*RouterEndpointUri*	camel-spring	Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint.
Timer	timer:*EndpointID*[?*Options*]	camel-core	A timer endpoint.
Validation	validator:*LocalOrRemoteResource*	camel-spring	Validates the payload of a message using XML Schema and JAXP Validation.
Velocity	velocity:*TemplateURI*[?*Options*]	camel-velocity	Generates a response using an Apache Velocity template.
VM	vm:*EndpointID*	camel-core	Used to deliver messages to a

[17] http://code.google.com/p/jsmpp/

Component	Endpoint URI	Artifact ID	Description
			java.util.concurrent.BlockingQueue, useful when creating SEDA style processing pipelines within the same JVM.
XMPP	`xmpp:`*`Hostname`*`[:`*`Port`*`][/`*`Room`*`]`	`camel-xmpp`	Working with XMPP and Jabber.
XQuery	`xquery:`*`TemplateURI`*	`camel-saxon`	Generates a response using an XQuery template.
XSLT	`xslt:`*`TemplateURI`*`[?`*`Options`*`]`	`camel-spring`	xquery:someXQueryResource.

Chapter 2. ActiveMQ

ActiveMQ Component

The ActiveMQ component allows messages to be sent to a JMS[1] Queue or Topic; or messages to be consumed from a JMS Queue or Topic using Apache ActiveMQ[2].

This component is based on the JMS Component on page 293 and uses Spring's JMS support for declarative transactions, using Spring's `JmsTemplate` for sending and a `MessageListenerContainer` for consuming. All the options from the JMS on page 293 component also apply for this component.

To use this component, make sure you have the `activemq.jar` or `activemq-core.jar` on your classpath along with any Fuse Mediation Router dependencies such as `camel-core.jar`, `camel-spring.jar` and `camel-jms.jar`.

URI format

```
activemq:[queue:|topic:]destinationName
```

Where **destinationName** is an ActiveMQ queue or topic name. By default, the **destinationName** is interpreted as a queue name. For example, to connect to the queue, `FOO.BAR`, use:

```
activemq:FOO.BAR
```

You can include the optional `queue:` prefix, if you prefer:

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

To connect to a topic, you must include the `topic:` prefix. For example, to connect to the topic, `Stocks.Prices`, use:

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

Options

See Options on the JMS on page 293 component as all these options also apply for this component.

[1] http://java.sun.com/products/jms/
[2] http://activemq.apache.org/

Configuring the Connection Factory

The following test case[3] shows how to add an ActiveMQComponent to the CamelContext using the `activeMQComponent()` method[4] while specifying the brokerURL[5] used to connect to ActiveMQ.

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

Configuring the Connection Factory using Spring XML

You can configure the ActiveMQ broker URL on the ActiveMQComponent as follows

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans-2.0.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <camelContext xmlns="http://camel.apache.org/schema/spring">
  </camelContext>


  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="tcp://somehost:61616"/>
  </bean>

</beans>
```

Using connection pooling

When sending to an ActiveMQ broker using Camel it's recommended to use a pooled connection factory to handle efficient pooling of JMS connections, sessions and producers. This is documented in the page ActiveMQ Spring Support[6].

You can grab Jencks AMQ pool with Maven:

[3] http://svn.apache.org/repos/asf/activemq/trunk/activemq-camel/src/test/java/org/apache/camel/component/ActiveMQRouteTest.java
[4] http://camel.apache.org/maven/current/camel-activemq/apidocs/org/apache/camel/component/activemq/ActiveMQComponent.html#activeMQComponent(java.lang.String)
[5] http://activemq.apache.org/configuring-transports.html
[6] http://activemq.apache.org/spring-support.html

```
    <dependency>
      <groupId>org.apache.activemq</groupId>
      <artifactId>activemq-pool</artifactId>
      <version>5.3.2</version>
    </dependency>
```

And then setup the **activemq** component as follows:

```
    <bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">

        <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>

   <bean id="pooledConnectionFactory" class="org.apache.activemq.pool.PooledConnectionFact
ory">
        <property name="maxConnections" value="8" />
        <property name="maximumActive" value="500" />
        <property name="connectionFactory" ref="jmsConnectionFactory" />
    </bean>


    <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
        <property name="connectionFactory" ref="pooledConnectionFactory"/>
        <property name="transacted" value="false"/>
        <property name="concurrentConsumers" value="10"/>
    </bean>

    <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
        <property name="configuration" ref="jmsConfig"/>
    </bean>
```

Invoking MessageListener POJOs in a route

The ActiveMQ component also provides a helper Type Converter[7] from a JMS MessageListener to a Processor. This means that the Bean on page 41 component is capable of invoking any JMS MessageListener bean directly inside any route.

So for example you can create a MessageListener in JMS as follows:

```
public class MyListener implements MessageListener {
   public void onMessage(Message jmsMessage) {
       // ...
   }
}
```

Then use it in your route as follows

[7] Type Converter

```
from("file://foo/bar").
  bean(MyListener.class);
```

That is, you can reuse any of the Fuse Mediation Router Components on page 3 and easily integrate them into your JMS `MessageListener` POJO\!

Consuming Advisory Messages

ActiveMQ can generate Advisory messages[8] which are put in topics that you can consume. Such messages can help you send alerts in case you detect slow consumers or to build statistics (number of messages/produced per day, etc.) The following Spring DSL example shows you how to read messages from a topic.

```
<route>
 <from uri="activemq:topic:ActiveMQ.Advisory.Connection?mapJmsMessage=false" />
 <convertBodyTo type="java.lang.String"/>
 <transform>
      <simple>${in.body}&#13;</simple>
 </transform>
 <to uri="file://data/activemq/?fileExist=Append&ileName=advisoryConnection-
${date:now:yyyyMMdd}.txt" />
</route>
```

If you consume a message on a queue, you should see the following files under data/activemq folder :

advisoryConnection-20100312.txt advisoryProducer-20100312.txt

and containing string:

```
     ActiveMQMessage {commandId = 0, responseRequired = false, messageId = ID:dell-charles-
3258-1268399815140
      -1:0:0:0:221, originalDestination = null, originalTransactionId = null, producerId =
 ID:dell-charles-
      3258-1268399815140-1:0:0:0, destination = topic://ActiveMQ.Advisory.Connection,
transactionId = null,
     expiration = 0, timestamp = 0, arrival = 0, brokerInTime = 1268403383468, brokerOutTime
 = 1268403383468,
      correlationId = null, replyTo = null, persistent = false, type = Advisory, priority
= 0, groupID = null,
     groupSequence = 0, targetConsumerId = null, compressed = false, userID = null, content
 = null,
      marshalledProperties = org.apache.activemq.util.ByteSequence@17e2705, dataStructure
= ConnectionInfo
      {commandId = 1, responseRequired = true, connectionId = ID:dell-charles-3258-
```

[8] http://activemq.apache.org/advisory-message.html

```
1268399815140-2:50,
      clientId = ID:dell-charles-3258-1268399815140-14:0, userName = , password = *****,
      brokerPath = null, brokerMasterConnector = false, manageable = true, clientMaster =
true},
      redeliveryCounter = 0, size = 0, properties = {originBrokerName=master, origin
BrokerId=ID:dell-charles-
      3258-1268399815140-0:0, originBrokerURL=vm://master}, readOnlyProperties = true,
readOnlyBody = true,
      droppable = false}
```

Getting Component JAR

You need these dependencies

- `camel-jms`
- `activemq-camel`

camel-jms

You **must** have the `camel-jms` as dependency as ActiveMQ on page 25 is an extension to the JMS on page 293 component.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jms</artifactId>
  <version>1.6.0</version>
</dependency>
```

The ActiveMQ component is released with the ActiveMQ project itself. For Maven 2 users you just need to add the following dependency to your project.

ActiveMQ 5.2 or later

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-camel</artifactId>
  <version>5.2.0</version>
</dependency>
```

ActiveMQ 5.1.0

For 5.1.0 its in the activemq-core library

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-core</artifactId>
  <version>5.1.0</version>
</dependency>
```

Alternatively you can download the component JAR file directly from the Maven repository:

- activemq-camel-5.2.0.jar[9]
- activemq-core-5.1.0.jar[10]

ActiveMQ 4.x

For this version you must use the JMS on page 293 component instead. Please be careful to use a pooling connection factory as described in the JmsTemplate Gotchas[11]

[9] http://repo2.maven.org/maven2/org/apache/activemq/activemq-camel/5.2.0/activemq-camel-5.2.0.jar
[10] http://repo2.maven.org/maven2/org/apache/activemq/activemq-core/5.1.0/activemq-core-5.1.0.jar
[11] http://activemq.apache.org/jmstemplate-gotchas.html

Chapter 3. ActiveMQ Journal

ActiveMQ Journal Component

The ActiveMQ Journal Component allows messages to be stored in a rolling log file and then consumed from that log file. The journal aggregates and batches up concurrent writes so that the overhead of writing and waiting for the disk sync is relatively constant, regardless of how many concurrent writes are being done. Therefore, this component supports and encourages you to use multiple concurrent producers to the same journal endpoint.

Each journal endpoint uses a different log file and therefore write batching (and the associated performance boost) does not occur between multiple endpoints.

This component only supports one active consumer on the endpoint. After the message is processed by the consumer's processor, the log file is marked and only subsequent messages in the log file will get delivered to consumers.

URI format

```
activemq.journal:directoryName[?options]
```

So for example, to send to the journal located in the `/tmp/data` directory you would use the following URI:

```
activemq.journal:/tmp/data
```

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

Options

Name	Default Value	Description
`syncConsume`	`false`	If set to `true`, when the journal is marked after a message is consumed, wait till the Operating System has verified the mark update is safely stored on disk.
`syncProduce`	`true`	If set to `true`, wait till the Operating System has verified the message is safely stored on disk.

Expected Exchange Data Types

The consumer of a Journal endpoint generates DefaultExchange[1] objects with the *In* message set as follows:

[1] http://activemq.apache.org/camel/maven/camel-core/apidocs/org/apache/camel/impl/DefaultExchange.html

- `journal` header: set to the endpoint URI of the journal the message came from.
- `location` header: set to a Location[2] which identifies where the record was stored on disk.
- Message body: set to ByteSequence[3], which contains the byte array data of the stored message.

The producer to a Journal endpoint expects an Exchange[4] with an *In* message where the body can be converted to a ByteSequence[5] or a `byte[]`.

[2] http://activemq.apache.org/maven/activemq-core/apidocs/org/apache/activemq/kaha/impl/async/Location.html
[3] http://activemq.apache.org/maven/activemq-core/apidocs/org/apache/activemq/util/ByteSequence.html
[4] http://activemq.apache.org/camel/maven/camel-core/apidocs/org/apache/camel/Exchange.html
[5] http://activemq.apache.org/maven/activemq-core/apidocs/org/apache/activemq/util/ByteSequence.html

Chapter 4. AMQP

AMQP

The AMQP component supports the AMQP protocol[1] via the Qpid[2] project.

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

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

URI format

```
amqp:[queue:|topic:]destinationName[?options]
```

You can specify all of the various configuration options of the JMS on page 293 component after the destination name.

[1] http://www.amqp.org/
[2] http://cwiki.apache.org/qpid/

Chapter 5. Atom

Atom Component

The **atom:** component is used for polling atom feeds.

Fuse Mediation Router will poll the feed every 60 seconds by default. **Note:** The component currently only supports polling (consuming) feeds.

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

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

URI format

```
atom://atomUri[?options]
```

Where **atomUri** is the URI to the Atom feed to poll.

Options

Property	Default	Description
`splitEntries`	`true`	If `true` Fuse Mediation Router will poll the feed and for the subsequent polls return each entry poll by poll. If the feed contains 7 entries then Fuse Mediation Router will return the first entry on the first poll, the 2nd entry on the next poll, until no more entries where as Fuse Mediation Router will do a new update on the feed. If `false` then Fuse Mediation Router will poll a fresh feed on every invocation.
`filter`	`true`	Is only used by the split entries to filter the entries to return. Fuse Mediation Router will default use the `UpdateDateFilter` that only return new entries from the feed. So the client consuming from the feed never receives the same entry more than once. The filter will return the entries ordered by the newest last.

`lastUpdate`	`null`	Is only used by the filter, as the starting timestamp for selection never entries (uses the `entry.updated` timestamp). Syntax format is: `yyyy-MM-ddTHH:MM:ss`. Example: `2007-12-24T17:45:59`.
`throttleEntries`	`true`	**Camel 2.5:** Sets whether all entries identified in a single feed poll should be delivered immediately. If `true`, only one entry is processed per `consumer.delay`. Only applicable when `splitEntries` is set to `true`.
`feedHeader`	`true`	Sets whether to add the Abdera Feed object as a header.
`sortEntries`	`false`	If `splitEntries` is `true`, this sets whether to sort those entries by updated date.
`consumer.delay`	`60000`	Delay in millis between each poll.
`consumer.initialDelay`	`1000`	Millis before polling starts.
`consumer.userFixedDelay`	`false`	If `true`, use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService[1] in JDK for details.

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

Exchange data format

Fuse Mediation Router will set the In body on the returned `Exchange` with the entries. Depending on the `splitEntries` flag Fuse Mediation Router will either return one `Entry` or a `List<Entry>`.

Option	Value	Behavior
`splitEntries`	`true`	Only a single entry from the currently being processed feed is set: `exchange.in.body(Entry)`
`splitEntries`	`false`	The entire list of entries from the feed is set: `exchange.in.body(List<Entry>)`

Fuse Mediation Router can set the `Feed` object on the in header (see `feedHeader` option to disable this):

Message Headers

Fuse Mediation Router atom uses these headers.

Header	Description
`org.apache.camel.component.atom.feed`	Fuse Mediation Router 1.x: When consuming the `org.apache.abdera.model.Feed` object is set to this header.

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html

`CamelAtomFeed`	Fuse Mediation Router 2.0: When consuming the `org.apache.abdera.model.Feed` object is set to this header.

Samples

In the following sample we poll James Strachan's blog:

```
from("atom://http://macstrac.blogspot.com/feeds/posts/default").to("seda:feeds");
```

In this sample we want to filter only good blogs we like to a SEDA queue. The sample also shows how to set up Fuse Mediation Router standalone, not running in any container or using Spring.

```
@Override
protected CamelContext createCamelContext() throws Exception {
    // First we register a blog service in our bean registry
    SimpleRegistry registry = new SimpleRegistry();
    registry.put("blogService", new BlogService());

    // Then we create the camel context with our bean registry
    context = new DefaultCamelContext(registry);

    // Then we add all the routes we need using the route builder DSL syntax
    context.addRoutes(createMyRoutes());

    // And finally we must start Camel to let the magic routing begins
    context.start();

    return context;
}

/**
 * This is the route builder where we create our routes using the Camel DSL syntax
 */
protected RouteBuilder createMyRoutes() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // We pool the atom feeds from the source for further processing in the seda
queue
            // we set the delay to 1 second for each pool as this is a unit test also and
we can
            // not wait the default poll interval of 60 seconds.
           // Using splitEntries=true will during polling only fetch one Atom Entry at any
 given time.
            // As the feed.atom file contains 7 entries, using this will require 7 polls
to fetch the entire
            // content. When Camel have reach the end of entries it will refresh the atom
feed from URI source
```

```
            // and restart - but as Camel by default uses the UpdatedDateFilter it will
only deliver new
            // blog entries to "seda:feeds". So only when James Straham updates his blog
with a new entry
            // Camel will create an exchange for the seda:feeds.
            from("atom:file:src/test/data/feed.atom?splitEntries=true&con
sumer.delay=1000").to("seda:feeds");

            // From the feeds we filter each blot entry by using our blog service class
            from("seda:feeds").filter().method("blogService", "isGoodBlog").to("seda:goodB
logs");

            // And the good blogs is moved to a mock queue as this sample is also used for
 unit testing
           // this is one of the strengths in Camel that you can also use the mock endpoint
 for your
            // unit tests
            from("seda:goodBlogs").to("mock:result");
        }
    };
}

/**
 * This is the actual junit test method that does the assertion that our routes is working

 * as expected
 */
@Test
public void testFiltering() throws Exception {
    // create and start Camel
    context = createCamelContext();
    context.start();

    // Get the mock endpoint
    MockEndpoint mock = context.getEndpoint("mock:result", MockEndpoint.class);

    // There should be at least two good blog entries from the feed
    mock.expectedMinimumMessageCount(2);

    // Asserts that the above expectations is true, will throw assertions exception if it
failed
   // Camel will default wait max 20 seconds for the assertions to be true, if the conditions

    // is true sooner Camel will continue
    mock.assertIsSatisfied();

    // stop Camel after use
    context.stop();
}
```

```
/**
 * Services for blogs
 */
public class BlogService {

    /**
     * Tests the blogs if its a good blog entry or not
     */
    public boolean isGoodBlog(Exchange exchange) {
        Entry entry = exchange.getIn().getBody(Entry.class);
        String title = entry.getTitle();

        // We like blogs about Camel
        boolean good = title.toLowerCase().contains("camel");
        return good;
    }

}
```

Chapter 6. Bean

Bean Component

The **bean:** component binds beans to Fuse Mediation Router message exchanges.

URI format

```
bean:beanID[?options]
```

Where **beanID** can be any string which is used to lookup look up the bean in the Registry

Options

Name	Type	Default	Description
`method`	`String`	`null`	The method name that bean will be invoked. If not provided, Fuse Mediation Router will try to pick the method itself. In case of ambiguity an exception is thrown. See Bean Binding for more details.
`cache`	`boolean`	`false`	If enabled, Fuse Mediation Router will cache the result of the first Registry look-up. Cache can be enabled if the bean in the Registry is defined as a singleton scope.
`multiParameterArray`	`boolean`	`false`	**Fuse Mediation Router 1.5:** How to treat the parameters which are passed from the message body; if it is `true`, the In message body should be an array of parameters.
`type`	`String`	`null`	**Fuse Mediation Router 2.6:** The fully qualified class name of the parameter type (or sub-type) of the method which should be called. This is only necessary, if you use method overloading and you have to tell Fuse Mediation Router which of the methods should be used. Otherwise you will fail with an `AmbiguousMethodCallException` exception.

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

Using

The object instance that is used to consume messages must be explicitly registered with the Registry. For example, if you are using Spring you must define the bean in the Spring configuration, `spring.xml`; or if you don't use Spring, put the bean in JNDI.

```
// lets populate the context with the services we need
// note that we could just use a spring.xml file to avoid this step
JndiContext context = new JndiContext();
context.bind("bye", new SayService("Good Bye!"));

CamelContext camelContext = new DefaultCamelContext(context);
```

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

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

A **bean:** endpoint cannot be defined as the input to the route; i.e. you cannot consume from it, you can only route from some inbound message Endpoint to the bean endpoint as output. So consider using a **direct:** or **queue:** endpoint as the input.

You can use the `createProxy()` methods on ProxyHelper[1] to create a proxy that will generate BeanExchanges and send them to any endpoint:

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

And the same route using Spring DSL:

```
    <route>
       <from uri="direct:hello">
       <to uri="bean:bye"/>
    </route>
```

Bean as endpoint

Fuse Mediation Router also supports invoking Bean on page 41 as an Endpoint. In the route below:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="myBean"/>
    <to uri="mock:results"/>
  </route>
</camelContext>
```

[1] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/bean/ProxyHelper.html

```
<bean id="myBean" class="org.apache.camel.spring.bind.ExampleBean"/>
```

What happens is that when the exchange is routed to the `myBean` Fuse Mediation Router will use the Bean Binding to invoke the bean. The source for the bean is just a plain POJO:

```
public class ExampleBean {

    public String sayHello(String name) {
        return "Hello " + name + "!";
    }
}
```

Fuse Mediation Router will use Bean Binding to invoke the `sayHello` method, by converting the Exchange's In body to the `String` type and storing the output of the method on the Exchange Out body.

Bean Binding

How bean methods to be invoked are chosen (if they are not specified explicitly through the **method** parameter) and how parameter values are constructed from the Message are all defined by the Bean Binding mechanism which is used throughout all of the various Bean Integration mechanisms in Fuse Mediation Router.

- Class on page 61 component
- Bean Binding
- Bean Integration

Chapter 7. Bean Validation

Bean Validation Component

Available as of 2.3

The Validation component performs bean validation of the message body using the Java Bean Validation API (JSR 303[1]). uses the reference implementation, which is Hibernate Validator[2].

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

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

URI format

```
bean-validator:something[?options]
```

or

```
bean-validator://something[?options]
```

Where **something** must be present to provide a valid URL. You can append query options to the URI in the following format, `?option=value&option=value&...`

URI Options

The following URI options are supported:

Option	Default	Description
`group`	`javax.validation.groups.Default`	The custom validation group to use.
`messageInterpolator`	`org.hibernate.validator.engine.ResourceBundleMessageInterpolator`	Reference to a custom `javax.validation.MessageInterp` Registry.

[1] http://jcp.org/en/jsr/detail?id=303
[2] http://docs.jboss.org/hibernate/stable/validator/reference/en/html_single/

`traversableResolver`	`org.hibernate.validator.engine.resolver.DefaultTraversableResolver`	Reference to a custom `javax.validation.TraversableReso` Registry.
`constraintValidatorFactory`	`org.hibernate.validator.engine.ConstraintValidatorFactoryImpl`	Reference to a custom `javax.validation.ConstraintValid` in the Registry.

Example

Assumed we have a java bean with the following annotations

Car.java

```
// Java
public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 5, max = 14, groups = OptionalChecks.class)
    private String licensePlate;

    // getter and setter
}
```

and an interface definition for our custom validation group

OptionalChecks.java

```
public interface OptionalChecks {
}
```

with the following route, only the **@NotNull** constraints on the attributes manufacturer and licensePlate will be validated (uses the default group `javax.validation.groups.Default`).

```
from("direct:start")
.to("bean-validator://x")
.to("mock:end")
```

If you want to check the constraints from the group `OptionalChecks`, you have to define the route like this

```
from("direct:start")
.to("bean-validator://x?group=OptionalChecks")
.to("mock:end")
```

If you want to check the constraints from both groups, you have to define a new interface first

AllChecks.java

```
@GroupSequence({Default.class, OptionalChecks.class})
public interface AllChecks {
}
```

and then your route definition should looks like this

```
from("direct:start")
.to("bean-validator://x?group=AllChecks")
.to("mock:end")
```

And if you have to provide your own message interpolator, traversable resolver and constraint validator factory, you have to write a route like this

```
<bean id="myMessageInterpolator" class="my.ConstraintValidatorFactory" />
<bean id="myTraversableResolver" class="my.TraversableResolver" />
<bean id="myConstraintValidatorFactory" class="my.ConstraintValidatorFactory" />

from("direct:start")
.to("bean-validator://x?group=AllChecks&messageInterpolator=#myMessageInterpolator&traversableResolver=#myTraversableResolver&constraintValidatorFactory=#myConstraintValidatorFactory")
.to("mock:end")
```

It's also possible to describe your constraints as XML and not as Java annotations. In this case, you have to provide the file `META-INF/validation.xml` which could looks like this

validation.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
 xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration">
 <default-provider>org.hibernate.validator.HibernateValidator</default-provider>
 <message-interpolator>org.hibernate.validator.engine.ResourceBundleMessageInterpolator</mes
sage-interpolator>
 <traversable-resolver>org.hibernate.validator.engine.resolver.DefaultTraversableResolv
er</traversable-resolver>
 <constraint-validator-factory>org.hibernate.validator.engine.ConstraintValidatorFactoryIm
pl</constraint-validator-factory>

 <constraint-mapping>/constraints-car.xml</constraint-mapping>
</validation-config>
```

and the `constraints-car.xml` file

constraints-car.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/mapping validation-mapping-
1.0.xsd"
 xmlns="http://jboss.org/xml/ns/javax/validation/mapping">
 <default-package>org.apache.camel.component.bean.validator</default-package>

 <bean class="CarWithoutAnnotations" ignore-annotations="true">
  <field name="manufacturer">
   <constraint annotation="javax.validation.constraints.NotNull" />
  </field>

  <field name="licensePlate">
   <constraint annotation="javax.validation.constraints.NotNull" />

   <constraint annotation="javax.validation.constraints.Size">
    <groups>
     <value>org.apache.camel.component.bean.validator.OptionalChecks</value>
    </groups>
    <element name="min">5</element>
    <element name="max">14</element>
   </constraint>
  </field>
 </bean>
</constraint-mappings>
```

Chapter 8. Browse

Browse Component

Available as of Fuse Mediation Router 2.0

The Browse component provides a simple BrowsableEndpoint which can be useful for testing, visualisation tools or debugging. The exchanges sent to the endpoint are all available to be browsed.

URI format

```
browse:someName
```

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

Sample

In the route below, we insert a `browse:` component to be able to browse the Exchanges that are passing through:

```
  from("activemq:order.in").to("browse:orderReceived").to("bean:processOrder");
```

We can now inspect the received exchanges from within the Java code:

```
    private CamelContext context;

    public void inspectRecievedOrders() {
        BrowsableEndpoint browse = context.getEndpoint("browse:orderReceived",
BrowsableEndpoint.class);
        List<Exchange> exchanges = browse.getExchanges();
        ...
        // then we can inspect the list of received exchanges from Java
        for (Exchange exchange : exchanges) {
            String payload = exchange.getIn().getBody();
            ...
        }
   }
```

Chapter 9. Cache

Cache Component

Available as of Fuse Mediation Router 2.1

The **cache** component enables you to perform caching operations using EHCache as the Cache Implementation. The cache itself is created on demand or if a cache of that name already exists then it is simply utilized with its original settings.

This component supports producer and event based consumer endpoints.

The Cache consumer is an event based consumer and can be used to listen and respond to specific cache activities. If you need to perform selections from a pre-existing cache, used the processors defined for the cache component.

URI format

```
cache://cacheName[?options]
```

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

Options

Name	Default Value	Description
`maxElementsInMemory`	`1000`	The numer of elements that may be stored in the defined cache
`memoryStoreEvictionPolicy`	`MemoryStoreEvictionPolicy.LFU`	The number of elements that may be stored in the defined cache. The policy options include: • `MemoryStoreEvictionPolicy.LFU`—Least frequently used. • `MemoryStoreEvictionPolicy.LRU`—Least recently used. • `MemoryStoreEvictionPolicy.FIFO`—First in first out, ordered by creation time.
`overflowToDisk`	`true`	Specifies whether cache may overflow to disk.

`eternal`	`false`	Sets whether elements are eternal. If eternal, timeouts are ignored and the element is never expired.
`timeToLiveSeconds`	`300`	The maximum time between creation time and when an element expires. Is only used if the element is not eternal.
`timeToIdleSeconds`	`300`	The maximum amount of time between accesses before an element expires.
`diskPersistent`	`true`	Whether the disk store persists between restarts of the Virtual Machine. The default value is `false`.
`diskExpiryThreadIntervalSeconds`	`120`	The number of seconds between runs of the disk expiry thread. The default value is 120 seconds.
`cacheManagerFactory`	`null`	**Camel 2.3:** If you want to use a custom factory which instantiates and creates the EHCache `net.sf.ehcache.CacheManager`.

Message Headers

Header	Description
`CACHE_OPERATION`	The operation to be performed on the cache. The valid options are: • `ADD` • `UPDATE` • `DELETE` • `DELETEALL`
`CACHE_KEY`	The cache key used to store the message in the cache. The cache key is optional, if the `CACHE_OPERATION` is `DELETEALL`.

Cache Producer

Sending data to the cache involves the ability to direct payloads in exchanges to be stored in a pre-existing or created-on- demand cache. The mechanics of doing this involve - setting the Message Exchange Headers shown above. - ensuring that the Message Exchange Body contains the message directed to the cache

Cache Consumer

Receiving data from the cache involves the ability of the `CacheConsumer` to listen on a pre-existing or created-on-demand Cache using an event Listener and receive automatic notifications when any cache activity take place (i.e `ADD`/`UPDATE`/`DELETE`/`DELETEALL`). Upon such an activity taking place - an exchange containing Message Exchange Headers and a Message Exchange Body containing the just added/updated payload is placed and sent. - in case of a `DELETEALL` operation, the Message Exchange Header `CACHE_KEY` and the Message Exchange Body are not populated.

Cache Processors

There are a set of nice processors with the ability to perform cache lookups and selectively replace payload content at the - body - token - xpath level

Example 1: Configuring the cache

```
from("cache://MyApplicationCache" +
          "?maxElementsInMemory=1000" +
          "&memoryStoreEvictionPolicy=" +
              "MemoryStoreEvictionPolicy.LFU" +
          "&overflowToDisk=true" +
          "&eternal=true" +
          "&timeToLiveSeconds=300" +
          "&timeToIdleSeconds=true" +
          "&diskPersistent=true" +
          "&diskExpiryThreadIntervalSeconds=300")
```

Example 2: Adding keys to the cache

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
     from("direct:start")
     .setHeader("CACHE_OPERATION", constant("ADD"))
     .setHeader("CACHE_KEY", constant("Ralph_Waldo_Emerson"))
     .to("cache://TestCache1")
   }
};
```

Example 2: Updating existing keys in a cache

```
RouteBuilder builder = new RouteBuilder() {
```

```
    public void configure() {
     from("direct:start")
     .setHeader("CACHE_OPERATION", constant("UPDATE"))
     .setHeader("CACHE_KEY", constant("Ralph_Waldo_Emerson"))
     .to("cache://TestCache1")
   }
};
```

Example 3: Deleting existing keys in a cache

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
     from("direct:start")
     .setHeader("CACHE_OPERATION", constant("DELETE"))
     .setHeader("CACHE_KEY", constant("Ralph_Waldo_Emerson"))
     .to("cache://TestCache1")
   }
};
```

Example 4: Deleting all existing keys in a cache

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
     from("direct:start")
     .setHeader("CACHE_OPERATION", constant("DELETEALL"))
     .to("cache://TestCache1");
    }
};
```

Example 5: Notifying any changes registering in a Cache to Processors and other Producers

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
     from("cache://TestCache1")
     .process(new Processor() {
        public void process(Exchange exchange)
               throws Exception {
           String operation = (String) exchange.getIn().getHeader("CACHE_OPERATION");
           String key = (String) exchange.getIn().getHeader("CACHE_KEY");
           Object body = exchange.getIn().getBody();
           // Do something
```

```
        }
      })
    }
};
```

Example 6: Using Processors to selectively replace payload with cache values

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
     //Message Body Replacer
     from("cache://TestCache1")
     .filter(header("CACHE_KEY").isEqualTo("greeting"))
     .process(new CacheBasedMessageBodyReplacer("cache://TestCache1","farewell"))
     .to("direct:next");

    //Message Token replacer
    from("cache://TestCache1")
    .filter(header("CACHE_KEY").isEqualTo("quote"))
    .process(new CacheBasedTokenReplacer("cache://TestCache1","novel","#novel#"))
    .process(new CacheBasedTokenReplacer("cache://TestCache1","author","#author#"))
    .process(new CacheBasedTokenReplacer("cache://TestCache1","number","#number#"))
    .to("direct:next");

    //Message XPath replacer
    from("cache://TestCache1").
    .filter(header("CACHE_KEY").isEqualTo("XML_FRAGMENT"))
    .process(new CacheBasedXPathReplacer("cache://TestCache1","book1","/books/book1"))
    .process (new CacheBasedXPathReplacer("cache://TestCache1","book2","/books/book2"))
    .to("direct:next");
   }
};
```

Example 7: Getting an entry from the Cache

```
from("direct:start")
    // Prepare headers
    .setHeader(CacheConstants.CACHE_OPERATION, constant(CacheConstants.CACHE_OPERATION_GET))

    .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson")).
    .to("cache://TestCache1").
    // Check if entry was not found
    .choice().when(header(CacheConstants.CACHE_ELEMENT_WAS_FOUND).isNull()).
        // If not found, get the payload and put it to cache
        .to("cxf:bean:someHeavyweightOperation").
        .setHeader(CacheConstants.CACHE_OPERATION, constant(CacheConstants.CACHE_OPERA
TION_ADD))
```

```
        .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
        .to("cache://TestCache1")
    .end()
    .to("direct:nextPhase");
```

Example 8: Checking for an entry in the Cache

Note: CHECK command tests existence of the entry in the cache but doesn't place message to the body.

```
from("direct:start")
    // Prepare headers
   .setHeader(CacheConstants.CACHE_OPERATION, constant(CacheConstants.CACHE_OPERATION_CHECK))

    .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson")).
    .to("cache://TestCache1").
    // Check if entry was not found
    .choice().when(header(CacheConstants.CACHE_ELEMENT_WAS_FOUND).isNull()).
        // If not found, get the payload and put it to cache
        .to("cxf:bean:someHeavyweightOperation").
        .setHeader(CacheConstants.CACHE_OPERATION, constant(CacheConstants.CACHE_OPERA
TION_ADD))
        .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
        .to("cache://TestCache1")
    .end();
```

Management of EHCache

EHCache has its own statistics and management from JMX[1].

Here's a snippet on how to expose them via JMX in a Spring application context:

```
     <bean id="ehCacheManagementService" class="net.sf.ehcache.management.ManagementService"
 init-method="init" lazy-init="false">
      <constructor-arg>
      <bean class="net.sf.ehcache.CacheManager" factory-method="getInstance"/>
      </constructor-arg>
      <constructor-arg>
     <bean class="org.springframework.jmx.support.JmxUtils" factory-method="locateMBeanServ
er"/>
      </constructor-arg>
      <constructor-arg value="true"/>
      <constructor-arg value="true"/>
      <constructor-arg value="true"/>
      <constructor-arg value="true"/>
```

[1] Camel JMX

```
    </bean>
```

Of course you can do the same thing in straight Java:

```
    ManagementService.registerMBeans(CacheManager.getInstance(), mbeanServer, true, true,
true, true);
```

You can get cache hits, misses, in-memory hits, disk hits, size stats this way. You can also change CacheConfiguration parameters on the fly.

Chapter 10. Class

Class Component

Available as of 2.4

The **class:** component binds beans to message exchanges. It works in the same way as the Bean on page 41 component but instead of looking up beans from a Registry it creates the bean based on the class name.

URI format

```
class:className[?options]
```

Where **className** is the fully qualified class name to create and use as bean.

Options

Name	Type	Default	Description
`method`	`String`	`null`	The method name that bean will be invoked. If not provided, will try to pick the method itself. In case of ambiguity an exception is thrown. See Bean Binding for more details.
`multiParameterArray`	`boolean`	`false`	How to treat the parameters which are passed from the message body; if it is `true`, the In message body should be an array of parameters.

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

Using

You simply use the **class** component just as the Bean on page 41 component but by specifying the fully qualified classname instead. For example to use the `MyFooBean` you have to do as follows:

```
   from("direct:start").to("class:org.apache.camel.component.bean.MyFooBean").to("mock:res
ult");
```

You can also specify which method to invoke on the `MyFooBean`, for example `hello`:

```
   from("direct:start").to("class:org.apache.camel.component.bean.MyFooBean?meth
```

```
od=hello").to("mock:result");
```

Setting properties on the created instance

In the endpoint uri you can specify properties to set on the created instance, for example if it has a `setPrefix` method:

```
    from("direct:start")
        .to("class:org.apache.camel.component.bean.MyPrefixBean?prefix=Bye")
        .to("mock:result");
```

And you can also use the # syntax to refer to properties to be looked up in the Registry.

```
    from("direct:start")
        .to("class:org.apache.camel.component.bean.MyPrefixBean?cool=#foo")
        .to("mock:result");
```

Which will lookup a bean from the Registry with the id `foo` and invoke the `setCool` method on the created instance of the `MyPrefixBean` class.

See more

See more details at the Bean on page 41 component as the **class** component works in much the same way.

- Bean on page 41
- Bean Binding
- Bean Integration

Chapter 11. Cometd

Cometd Component

The **cometd:** component is a transport for working with the jetty[1] implementation of the cometd/bayeux protocol[2]. Using this component in combination with the dojo toolkit library it's possible to push Fuse Mediation Router messages directly into the browser using an AJAX based mechanism.

URI format

```
cometd://host:port/channelName[?options]
```

The **channelName** represents a topic that can be subscribed to by the Fuse Mediation Router endpoints.

Examples

```
cometd://localhost:8080/service/mychannel
cometds://localhost:8443/service/mychannel
```

where `cometds:` represents an SSL configured endpoint.

Options

Name	Default Value	Description
`resourceBase`		The root directory for the web resources or classpath. Use the protocol file: or classpath: depending if you want that the component loads the resource from file system or classpath. Classpath is required for OSGI deployment where the resources are packaged in the jar
`timeout`	`240000`	The server side poll timeout in milliseconds. This is how long the server will hold a reconnect request before responding.
`interval`	`0`	The client side poll timeout in milliseconds. How long a client will wait between reconnects
`maxInterval`	`30000`	The max client side poll timeout in milliseconds. A client will be removed if a connection is not received in this time.
`multiFrameInterval`	`1500`	The client side poll timeout, if multiple connections are detected from the same browser.

[1] http://www.mortbay.org/jetty
[2] http://docs.codehaus.org/display/JETTY/Cometd+%28aka+Bayeux%29

`jsonCommented`	`true`	If `true`, the server will accept JSON wrapped in a comment and will generate JSON wrapped in a comment. This is a defence against Ajax Hijacking.
`logLevel`	`1`	`0`=none, `1`=info, `2`=debug.

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

Here is some examples of how to pass the parameters.

For file (when the Webapp resources are located in the Web Application directory) `cometd://localhost:8080?resourceBase=file./webapp`. For classpath (when the web resources are packaged inside the Webapp folder) `cometd://localhost:8080?resourceBase=classpath:webapp`.

Chapter 12. Crypto (Digital Signatures)

Crypto component for Digital Signatures

Available as of Fuse Mediation Router 2.3

Using Fuse Mediation Router cryptographic endpoints and Java's Cryptographic extension it is easy to create Digital Signatures for Exchanges. Fuse Mediation Router provides a pair of flexible endpoints which get used in concert to create a signature for an exchange in one part of the exchange's workflow and then verify the signature in a later part of the workflow.

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

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

Introduction

Digital signatures make use Asymmetric Cryptographic techniques to sign messages. From a (very) high level, the algorithms use pairs of complimentary keys with the special property that data encrypted with one key can only be decrypted with the other. One, the private key, is closely guarded and used to 'sign' the message while the other, public key, is shared around to anyone interested in verifying your messages. Messages are signed by encrypting a digest of the message with the private key. This encrypted digest is transmitted along with the message. On the other side the verifier recalculates the message digest and uses the public key to decrypt the the digest in the signature. If both digest match the verifier knows only the holder of the private key could have created the signature.

Fuse Mediation Router uses the Signature service from the Java Cryptographic Extension to do all the heavy cryptographic lifting required to create exchange signatures. The following are some excellent sources for explaining the mechanics of Cryptography, Message digests and Digital Signatures and how to leverage them with the JCE.

- Bruce Schneier's Applied Cryptography
- Beginning Cryptography with Java by David Hook

- The ever insightful, Wikipedia Digital_signatures[1]

URI format

As mentioned Fuse Mediation Router provides a pair of crypto endpoints to create and verify signatures

```
crypto:sign:name[?options]
crypto:verify:name[?options]
```

- `crypto:sign` creates the signature and stores it in the Header keyed by the constant `Exchange.SIGNATURE`, i.e. `"CamelDigitalSignature"`.
- `crypto:verify` will read in the contents of this header and do the verification calculation.

In order to correctly function, sign and verify need to share a pair of keys, sign requiring a `PrivateKey` and verify a `PublicKey` (or a `Certificate` containing one). Using the JCE is is very simple to generate these key pairs but it is usually most secure to use a KeyStore to house and share your keys. The DSL is very flexible about how keys are supplied and provides a number of mechanisms.

Note a `crypto:sign` endpoint is typically defined in one route and the complimentary `crypto:verify` in another, though for simplicity in the examples they appear one after the other. It goes without saying that both sign and verify should be configured identically.

Options

Name	Type	Default	Description
`algorithm`	`String`	`DSA`	The name of the JCE Signature algorithm that will be used.
`alias`	`String`	`null`	An alias name that will be used to select a key from the keystore.
`bufferSize`	`Integer`	`2048`	the size of the buffer used in the signature process.
`certificate`	`Certificate`	`null`	A Certificate used to verify the signature of the exchange's payload. Either this or a Public Key is required.
`keystore`	`KeyStore`	`null`	A reference to a JCE Keystore that stores keys and certificates used to sign and verify.
`provider`	`String`	`null`	The name of the JCE Security Provider that should be used.
`privateKey`	`PrivatKey`	`null`	The private key used to sign the exchange's payload.
`publicKey`	`PublicKey`	`null`	The public key used to verify the signature of the exchange's payload.

[1] http://en.wikipedia.org/wiki/Digital_signature

`secureRandom`	`secureRandom`	`null`	A reference to a `SecureRandom` object that wil lbe used to initialize the Signature service.
`password`	`char[]`	`null`	The password for the keystore.

1) Raw keys

The most basic way to way to sign an verify an exchange is with a KeyPair as follows.

```
from("direct:keypair").to("crypto:sign://basic?privateKey=#myPrivateKey", "crypto:verify://ba
sic?publicKey=#myPublicKey", "mock:result");
```

The same can be achieved with the Spring XML Extensions[2] using references to keys

```
<route>
    <from uri="direct:keypair"/>
    <to uri="crypto:sign://basic?privateKey=#myPrivateKey" />
    <to uri="crypto:verify://basic?publicKey=#myPublicKey" />
    <to uri="mock:result"/>
</route>
```

2) KeyStores and Aliases.

The JCE provides a very versatile KeyStore for housing pairs of PrivateKeys and Certificates keeping them encrypted and password protected. They can be retrieved from it by applying an alias to the retrieval apis. There are a number of ways to get keys and Certificates into a keystore most often this is done with the external 'keytool' application. This[3] is a good example of using keytool to create a KeyStore with a self signed Cert and Private key.

The examples use a Keystore with a key and cert aliased by 'bob'. The password for the keystore and the key is 'letmein'

The following shows how to use a Keystore via the Fluent builders, it also shows how to load and initialize the keystore.

```
from("direct:keystore").to("crypto:sign://keystore?keystore=#keystore&alias=bob&password=let
mein", "crypto:verify://keystore?keystore=#keystore&alias=bob", "mock:result");
```

Again in Spring a ref is used to lookup an actual keystore instance.

```
<route>
    <from uri="direct:keystore"/>
    <to uri="crypto:sign://keystore?keystore=#keystore&lias=bob&assword=letmein" />
```

[2] Spring XML Extensions
[3] http://www.exampledepot.com/egs/java.security.cert/CreateCert.html

```
    <to uri="crypto:verify://keystore?keystore=#keystore&lias=bob" />
    <to uri="mock:result"/>
</route>
```

3) Changing JCE Provider and Algorithm

Changing the Signature algorithm or the Security provider is a simple matter of specifying their names. You will need to also use Keys that are compatible with the algorithm you choose.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(512, new SecureRandom());
keyPair = keyGen.generateKeyPair();
PrivateKey privateKey = keyPair.getPrivate();
PublicKey publicKey = keyPair.getPublic();

// we can set the keys explicitly on the endpoint instances.
context.getEndpoint("crypto:sign://rsa?algorithm=MD5withRSA", DigitalSignatureEnd
point.class).setPrivateKey(privateKey);
context.getEndpoint("crypto:verify://rsa?algorithm=MD5withRSA", DigitalSignatureEnd
point.class).setPublicKey(publicKey);
from("direct:algorithm").to("crypto:sign://rsa?algorithm=MD5withRSA", "crypto:verify://rsa?al
gorithm=MD5withRSA", "mock:result");
```

```
from("direct:provider").to("crypto:sign://provider?privateKey=#myPrivateKey&provider=SUN",
 "crypto:verify://provider?publicKey=#myPublicKey&provider=SUN", "mock:result");
```

or

```
<route>
    <from uri="direct:algorithm"/>
    <to uri="crypto:sign://rsa?algorithm=MD5withRSA&rivateKey=#rsaPrivateKey" />
    <to uri="crypto:verify://rsa?algorithm=MD5withRSA&ublicKey=#rsaPublicKey" />
    <to uri="mock:result"/>
</route>
```

```
<route>
    <from uri="direct:provider"/>
    <to uri="crypto:sign://provider?privateKey=#myPrivateKey&rovider=SUN" />
    <to uri="crypto:verify://provider?publicKey=#myPublicKey&rovider=SUN" />
    <to uri="mock:result"/>
</route>
```

4) Changing the Signature Mesasge Header

It may be desirable to change the message header used to store the signature. A different header name can be specified in the route definition as follows

```
from("direct:signature-header").to("crypto:sign://another?privateKey=#myPrivateKey&signa
tureHeader=AnotherDigitalSignature",
                                   "crypto:verify://another?publicKey=#myPublicKey&signa
tureHeader=AnotherDigitalSignature", "mock:result");
```

or

```
<route>
    <from uri="direct:signature-header"/>
   <to uri="crypto:sign://another?privateKey=#myPrivateKey&ignatureHeader=AnotherDigitalSig
nature" />
   <to uri="crypto:verify://another?publicKey=#myPublicKey&ignatureHeader=AnotherDigitalSig
nature" />
    <to uri="mock:result"/>
</route>
```

5) Changing the buffersize

In case you need to update the size of the buffer...

```
from("direct:buffersize").to("crypto:sign://buffer?privateKey=#myPrivateKey&buffersize=1024",
 "crypto:verify://buffer?publicKey=#myPublicKey&buffersize=1024", "mock:result");
```

or

```
<route>
    <from uri="direct:buffersize" />
    <to uri="crypto:sign://buffer?privateKey=#myPrivateKey&uffersize=1024" />
    <to uri="crypto:verify://buffer?publicKey=#myPublicKey&uffersize=1024" />
    <to uri="mock:result"/>
</route>
```

6) Supplying Keys dynamically.

When using a Recipient list or similar EIP the recipient of an exchange can vary dynamically. Using the same key across all recipients may neither be feasible or desirable. It would be useful to be able to specify the signature keys dynamically on a per exchange basis. The exchange could then be dynamically enriched with the key of its target recipient prior to signing. To facilitate this the signature mechanisms allow for keys to be supplied dynamically via the message headers below

- `Exchange.SIGNATURE_PRIVATE_KEY`, `"CamelSignaturePrivateKey"`
- `Exchange.SIGNATURE_PUBLIC_KEY_OR_CERT`, `"CamelSignaturePublicKeyOrCert"`

```
from("direct:headerkey-sign").to("crypto:sign://alias");
from("direct:headerkey-verify").to("crypto:verify://alias", "mock:result");
```

or

```
<route>
    <from uri="direct:headerkey-sign"/>
    <to uri="crypto:sign://headerkey" />
</route>
<route>
    <from uri="direct:headerkey-verify"/>
    <to uri="crypto:verify://headerkey" />
    <to uri="mock:result"/>
</route>
```

Better again would be to dynamically supply a keystore alias. Again the alias can be supplied in a message header

- `Exchange.KEYSTORE_ALIAS`, `"CamelSignatureKeyStoreAlias"`

```
from("direct:alias-sign").to("crypto:sign://alias?keystore=#keystore");
from("direct:alias-verify").to("crypto:verify://alias?keystore=#keystore", "mock:result");
```

or

```
<route>
    <from uri="direct:alias-sign"/>
    <to uri="crypto:sign://alias?keystore=#keystore" />
</route>
<route>
    <from uri="direct:alias-verify"/>
    <to uri="crypto:verify://alias?keystore=#keystore" />
    <to uri="mock:result"/>
</route>
```

The header would be set as follows

```
Exchange unsigned = getMandatoryEndpoint("direct:alias-sign").createExchange();
unsigned.getIn().setBody(payload);
unsigned.getIn().setHeader(DigitalSignatureConstants.KEYSTORE_ALIAS, "bob");
unsigned.getIn().setHeader(DigitalSignatureConstants.KEYSTORE_PASSWORD, "letmein".toCharAr
ray());
template.send("direct:alias-sign", unsigned);
Exchange signed = getMandatoryEndpoint("direct:alias-sign").createExchange();
signed.getIn().copyFrom(unsigned.getOut());
signed.getIn().setHeader(KEYSTORE_ALIAS, "bob");
template.send("direct:alias-verify", signed);
```

See also:

- Crypto Crypto is also available as a Data Format

Chapter 13. CXF Bean Component

CXF Bean Component (2.0 or later)

The **cxfbean:** component allows other Camel endpoints to send exchange and invoke Web service bean objects. (**Currently, it only supports JAXRS, JAXWS(new to camel2.1) annotated service bean.**)

Note: `CxfBeanEndpoint` is a `ProcessorEndpoint` so it has no consumers. It works similarly to a Bean component.

URI format

```
cxfbean:serviceBeanRef
```

Where **serviceBeanRef** is a registry key to look up the service bean object. If `serviceBeanRef` references a `List` object, elements of the `List` are the service bean objects accepted by the endpoint.

Options

Name	Description	Example
`cxfBeanBinding`	CXF bean binding specified by the `\#` notation. The referenced object must be an instance of `org.apache.camel.component.cxf.cxfbean.CxfBeanBinding`.	`cxfBinding=#bindingNam`
`bus`	CXF bus reference specified by the `\#` notation. The referenced object must be an instance of `org.apache.cxf.Bus`.	`bus=#busName`
`headerFilterStrategy`	Header filter strategy specified by the `\#` notation. The referenced object must be an instance of `org.apache.camel.spi.HeaderFilterStrategy`.	`headerFilterStrategy=#`
`setDefaultBus`	Will set the default bus when CXF endpoint create a bus by itself.	`true`, `false`
`populateFromClass`	Since 2.3, the wsdlLocation annotated in the POJO is ignored (by default) unless this option is set to `false`. Prior to 2.3, the wsdlLocation annotated in the POJO is always honored and it is not possible to ignore.	`true`, `false`
`providers`	Since 2.5, setting the providers for the CXFRS endpoint.	`providers=#providerRef`

Headers

Name	Description	Type	Required?	Default Value	In/Out	Examples
`CamelHttpCharacterEncoding` (before 2.0-m2: `CamelCxfBeanCharacterEncoding`)	Character encoding	`String`	No	None	In	ISO-8859-1
`CamelContentType` (before 2.0-m2: `CamelCxfBeanContentType`)	Content type	`String`	No	**/**	In	`text/xml`
CamelHttpBaseUri (2.0-m3 and before: `CamelCxfBeanRequestBasePath`)	The value of this header will be set in the CXF message as the `Message.BASE_PATH` property. It is needed by CXF JAX-RS processing. Basically, it is the scheme, host and port portion of the request URI.	`String`	Yes	The Endpoint URI of the source endpoint in the Camel exchange	In	http://localhost:9000 [1]
`CamelHttpPath` (before 2.0-m2: `CamelCxfBeanRequestPat{}h`)	Request URI's path	`String`	Yes	None	In	`consumer/123`
`CamelHttpMethod` (before 2.0-m2: `CamelCxfBeanVerb`)	RESTful request verb	`String`	Yes	None	In	`GET`, `PUT`, `POST`, `DELETE`
`CamelHttpResponseCode`	HTTP response code	`Integer`	No	None	Out	200

Note: Currently, CXF Bean component has (only) been tested with Jetty HTTP component it can understand headers from Jetty HTTP component without requiring conversion.

A Working Sample

This sample shows how to create a route that starts a Jetty HTTP server. The route sends requests to a CXF Bean and invokes a JAXRS annotated service.

First, create a route as follows. The `from` endpoint is a Jetty HTTP endpoint that is listening on port 9000. Notice that the `matchOnUriPrefix` option must be set to `true` because RESTful request URI will not match the endpoint's URI http://localhost:9000 exactly.

[1] http://localhost:9000

```
<route>
 <from uri="jetty:http://localhost:9000?matchOnUriPrefix=true" />
 <to uri="cxfbean:customerServiceBean" />
</route>
```

The `to` endpoint is a CXF Bean with bean name `customerServiceBean`. The name will be looked up from the registry. Next, we make sure our service bean is available in Spring registry. We create a bean definition in the Spring configuration. In this example, we create a List of service beans (of one element). We could have created just a single bean without a List.

```
<util:list id="customerServiceBean">
 <bean class="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService" />
</util:list>

<bean class="org.apache.camel.wsdl_first.PersonImpl" id="jaxwsBean" />
```

That's it. Once the route is started, the web service is ready for business. A HTTP client can make a request and receive response.

```
url = new URL("http://localhost:9000/customerservice/orders/223/products/323");
in = url.openStream();
assertEquals("{\"Product\":{\"description\":\"product 323\",\"id\":323}}", CxfUtils.get
StringFromInputStream(in));
```

Chapter 14. CXF

CXF Component

The **cxf:** component provides integration with Apache CXF[1] for connecting to JAX-WS services hosted in CXF.

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

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

CXF dependencies

If you want to learn about CXF dependencies, see the `WHICH-JARS`[2] text file.

Note

When using CXF as a consumer, the CAMEL:CXF Bean Component allows you to factor out how message payloads are received from their processing as a RESTful or SOAP web service. This has the potential of using a multitude of transports to consume web services. The bean component's configuration is also simpler and provides the fastest method to implement web services using Camel and CXF.

URI format

```
cxf:bean:cxfEndpoint[?options]
```

Where **cxfEndpoint** represents a bean ID that references a bean in the Spring bean registry. With this URI format, most of the endpoint details are specified in the bean definition.

```
cxf://someAddress[?options]
```

[1] http://incubator.apache.org/cxf/
[2] http://svn.apache.org/repos/asf/cxf/trunk/distribution/src/main/release/lib/WHICH_JARS

Where **someAddress** specifies the CXF endpoint's address. With this URI format, most of the endpoint details are specified using options.

For either style above, you can append options to the URI as follows:

```
cxf:bean:cxfEndpoint?wsdlURL=wsdl/hello_world.wsdl&dataFormat=PAYLOAD
```

Options

Name	Description	Sample Values
`wsdlURL`	The location of the WSDL.	file://local/wsdl/hello.wsdl or `wsdl/hel`
`serviceClass`	The name of the SEI (Service Endpoint Interface) class. This class can have, but does not require, JSR181 annotations. **Since 2.0,** this option is only required by POJO mode. If the wsdlURL option is provided, serviceClass is not required for PAYLOAD and MESSAGE mode. When wsdlURL option is used without serviceClass, the serviceName and portName (endpointName for Spring configuration) options **MUST** be provided. Since 2.0, it is possible to use `#` notation to reference a `serviceClass` object instance from the registry.. **Please be advised that the referenced object cannot be a Proxy (Spring AOP Proxy is OK)** as it relies on `Object.getClass().getName()` method for non Spring AOP Proxy.	`org.apache.camel.Hello`
`serviceClassInstance`	**In 1.6 or later (will be deprecated in 2.0)**, `serviceClassInstance` works like `serviceClass=#beanName`, which looks up a `serviceObject` instance from the registry.	`serviceClassInstance=beanName`
`serviceName`	The service name this service is implementing, it maps to the `wsdl:service@name`.	`{http://org.apache.camel}Service`
`portName`	The port name this service is implementing, it maps to the `wsdl:port@name`.	`{http://org.apache.camel}PortNam`

`dataFormat`	Which message data format the CXF endpoint supports	`POJO`, `PAYLOAD`, `MESSAGE`
`relayHeaders`	Available since Fuse Mediation Router 1.6.1. Please see the **Description of relayHeader option** section for this option in Fuse Mediation Router 2.0. Should a CXF endpoint relay headers along the route. Currently only available when `dataFormat=POJO`	`true`, `false`
`wrapped`	Which kind of operation that the CXF endpoint producer will invoke.	`true`, `false`
`wrappedStyle`	**New in 2.5.0** The WSDL style that describes how parameters are represented in the SOAP body. If the value is false, CXF will chose the document-literal unwrapped style, If the value is true, CXF will chose the document-literal wrapped style	`true`, `false`
`setDefaultBus`	Specifies whether or not to use the default CXF bus for this endpoint.	`true`, `false`
`bus`	New in Fuse Mediation Router 2.0, use `#` notation to reference a bus object from the registry. The referenced object must be an instance of `org.apache.cxf.Bus`.	`bus=#busName`
`cxfBinding`	New in Fuse Mediation Router 2.0, use `#` notation to reference a CXF binding object from the registry. The referenced object must be an instance of `org.apache.camel.component.cxf.CxfBinding`.	`cxfBinding=#bindingName`
`headerFilterStrategy`	New in Fuse Mediation Router 2.0, use `#` notation to reference a header filter strategy object from the registry. The referenced object must be an instance of `org.apache.camel.spi.HeaderFilterStrategy`.	`headerFilterStrategy=#strateg`
`loggingFeatureEnabled`	New in 2.3, this option enables CXF Logging Feature which writes inbound and outbound SOAP messages to log.	`loggingFeatureEnabled{}=true`
`defaultOperationName`	New in 2.4, this option will set the default operationName that will be used by the CxfProducer which invokes the remote service.	`defaultOperationName{}=greetM`
`defaultOperationNameSpace`	New in 2.4, this option will set the default operationNamespace that will be used by the CxfProducer which invokes the remote service.	`defaultOperationNamespace{}= http://apache.org/hello_world`
`synchronous`	New in 2.5, this option will let cxf endpoint decide to use sync or async API to do the underlying work.	`synchronous=true`

	The default value is false which means camel-cxf endpoint will try to use async API by default.
`publishedEndpointUrl`	New in 2.5, this option can override the endpointUrl that published from the WSDL which can be accessed with service address url plus ?wsdl. `publshedEndpointUrl=http://examp`

The `serviceName` and `portName` are QNames[3], so if you provide them be sure to prefix them with their `{namespace}` as shown in the examples above.

NOTE From Fuse Mediation Router 1.5.1, the `serviceClass` for a CXF producer (that is, the `to` endpoint) should be a Java interface.

The descriptions of the dataformats

DataFormat	Description
`POJO`	POJOs (plain old Java objects) are the Java parameters to the method being invoked on the target server. Both Protocol and Logical JAX-WS handlers are supported.
`PAYLOAD`	`PAYLOAD` is the message payload (the contents of the `soap:body`) after message configuration in the CXF endpoint is applied. Only Protocol JAX-WS handler is supported. Logical JAX-WS handler is not supported.
`MESSAGE`	`MESSAGE` is the raw message that is received from the transport layer. JAX-WS handler is not supported.

You can determine the data format mode of an exchange by retrieving the exchange property, `CamelCXFDataFormat`. The exchange key constant is defined in `org.apache.camel.component.cxf.CxfConstants.DATA_FORMAT_PROPERTY`.

How to enable CXF's LoggingOutInterceptor in MESSAGE mode

CXF's `LoggingOutInterceptor` outputs outbound message that goes on the wire to logging system (`java.util.logging`). Since the `LoggingOutInterceptor` is in `PRE_STREAM` phase (but `PRE_STREAM` phase is removed in `MESSAGE` mode), you have to configure `LoggingOutInterceptor` to be run during the `WRITE` phase. The following is an example.

```
   <bean id="loggingOutInterceptor" class="org.apache.cxf.interceptor.LoggingOutInterceptor">

         <!--  it really should have been user-prestream but CXF does have such phase! -->
         <constructor-arg value="write"/>
   </bean>

<cxf:cxfEndpoint id="serviceEndpoint" address="http://localhost:9002/helloworld"
```

[3] http://en.wikipedia.org/wiki/QName

```
 serviceClass="org.apache.camel.component.cxf.HelloService">
 <cxf:outInterceptors>
     <ref bean="loggingOutInterceptor"/>
 </cxf:outInterceptors>
 <cxf:properties>
  <entry key="dataFormat" value="MESSAGE"/>
 </cxf:properties>
</cxf:cxfEndpoint>
```

Description of relayHeaders option

There are *in-band* and *out-of-band* on-the-wire headers from the perspective of a JAXWS WSDL-first developer.

The *in-band* headers are headers that are explicitly defined as part of the WSDL binding contract for an endpoint such as SOAP headers.

The *out-of-band* headers are headers that are serialized over the wire, but are not explicitly part of the WSDL binding contract.

Headers relaying/filtering is bi-directional.

When a route has a CXF endpoint and the developer needs to have on-the-wire headers, such as SOAP headers, be relayed along the route to be consumed say by another JAXWS endpoint, then `relayHeaders` should be set to `true`, which is the default value.

Available in Release 1.6.1 and after (only in POJO mode)

The `relayHeaders=true` setting expresses an intent to relay the headers. The actual decision on whether a given header is relayed is delegated to a pluggable instance that implements the `MessageHeadersRelay` interface. A concrete implementation of `MessageHeadersRelay` will be consulted to decide if a header needs to be relayed or not. There is already an implementation of `SoapMessageHeadersRelay` which binds itself to well-known SOAP name spaces. Currently only out-of-band headers are filtered, and in-band headers will always be relayed when `relayHeaders=true`. If there is a header on the wire, whose name space is unknown to the runtime, then a fall back `DefaultMessageHeadersRelay` will be used, which simply allows all headers to be relayed.

The `relayHeaders=false` setting asserts that all headers, in-band and out-of-band, will be dropped.

You can plugin your own `MessageHeadersRelay` implementations overriding or adding additional ones to the list of relays. In order to override a preloaded relay instance just make sure that your `MessageHeadersRelay` implementation services the same name spaces as the one you looking to override. Also note, that the overriding relay has to service all of the name spaces as the one you looking to override, or else a runtime exception on route start up will be thrown as this would introduce an ambiguity in name spaces to relay instance mappings.

```
<cxf:cxfEndpoint ...>
   <cxf:properties>
     <entry key="org.apache.camel.cxf.message.headers.relays">
       <list>
         <ref bean="customHeadersRelay"/>
       </list>
     </entry>
   </cxf:properties>
 </cxf:cxfEndpoint>
 <bean id="customHeadersRelay" class="org.apache.camel.component.cxf.soap.headers.Custom
HeadersRelay"/>
```

Take a look at the tests that show how you'd be able to relay/drop headers here:

https://svn.apache.org/repos/asf/camel/branches/camel-1.x/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTest.java
[4]

Changes since Release 2.0

- `POJO` and `PAYLOAD` modes are supported. In `POJO` mode, only out-of-band message headers are available for filtering as the in-band headers have been processed and removed from the header list by CXF. The in-band headers are incorporated into the `MessageContentList` in `POJO` mode. The `camel-cxf` component does make any attempt to remove the in-band headers from the `MessageContentList` as it does in 1.6.1. If filtering of in-band headers is required, please use `PAYLOAD` mode or plug in a (pretty straightforward) CXF interceptor/JAXWS Handler to the CXF endpoint.

- The Message Header Relay mechanism has been merged into `CxfHeaderFilterStrategy`. The `relayHeaders` option, its semantics, and default value remain the same, but it is a property of `CxfHeaderFilterStrategy`.Here is an example of configuring it.

  ```
  <bean id="dropAllMessageHeadersStrategy" class="org.apache.camel.component.cxf.CxfHeader
  FilterStrategy">

      <!--  Set relayHeaders to false to drop all SOAP headers -->
      <property name="relayHeaders" value="false"/>

  </bean>
  ```

  Then, your endpoint can reference the `CxfHeaderFilterStrategy`.

  ```
  <route>
      <from uri="cxf:bean:routerNoRelayEndpoint?headerFilterStrategy=#dropAllMessageHead
  ersStrategy"/>
      <to uri="cxf:bean:serviceNoRelayEndpoint?headerFilterStrategy=#dropAllMessageHead
  ```

[4] https://svn.apache.org/repos/asf/camel/branches/camel-1.x/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTest.java

```
ersStrategy"/>
</route>
```

- The `MessageHeadersRelay` interface has changed slightly and has been renamed to `MessageHeaderFilter`. It is a property of `CxfHeaderFilterStrategy`. Here is an example of configuring user defined Message Header Filters:

```
<bean id="customMessageFilterStrategy" class="org.apache.camel.component.cxf.CxfHeaderFil
terStrategy">
    <property name="messageHeaderFilters">
        <list>
            <!--  SoapMessageHeaderFilter is the built in filter.  It can be removed by
omitting it. -->
            <bean class="org.apache.camel.component.cxf.SoapMessageHeaderFilter"/>


            <!--  Add custom filter here -->
          <bean class="org.apache.camel.component.cxf.soap.headers.CustomHeaderFilter"/>

        </list>
    </property>
</bean>
```

- Other than `relayHeaders`, there are new properties that can be configured in `CxfHeaderFilterStrategy`.

Name	Description	type	Required?	Default value
`relayHeaders`	All message headers will be processed by Message Header Filters	`boolean`	No	`true` (1.6.1 behavior)
`relayAllMessageHeaders`	All message headers will be propagated (without processing by Message Header Filters)	`boolean`	No	`false` (1.6.1 behavior)
`allowFilterNamespaceClash`	If two filters overlap in activation namespace, the property control how it should be handled. If the value is `true`, last one wins. If the value is `false`, it will throw an exception	`boolean`	No	`false` (1.6.1 behavior)

Configure the CXF endpoints with Spring

You can configure the CXF endpoint with the Spring configuration file shown below, and you can also embed the endpoint into the `camelContext` tags. When you are invoking the service endpoint, you can set the `operationName` and `operationNameSpace` headers to explicitly state which operation you are calling.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://activemq.apache.org/camel/schema/cxfEndpoint"
       xsi:schemaLocation="
              http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans-2.0.xsd
              http://activemq.apache.org/camel/schema/cxfEndpoint http://act
ivemq.apache.org/camel/schema/cxf/camel-cxf-1.6.0.xsd
              http://activemq.apache.org/camel/schema/spring http://act
ivemq.apache.org/camel/schema/spring/camel-spring.xsd     ">
    <cxf:cxfEndpoint id="routerEndpoint" address="http://localhost:9003/CamelContext/Router
Port"
      serviceClass="org.apache.hello_world_soap_http.GreeterImpl"/>
     <cxf:cxfEndpoint id="serviceEndpoint" address="http://localhost:9000/SoapContext/Soap
Port"
      wsdlURL="testutils/hello_world.wsdl"
      serviceClass="org.apache.hello_world_soap_http.Greeter"
      endpointName="s:SoapPort"
      serviceName="s:SOAPService"
     xmlns:s="http://apache.org/hello_world_soap_http" />
    <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
      <route>
        <from uri="cxf:bean:routerEndpoint" />
        <to uri="cxf:bean:serviceEndpoint" />
      </route>
    </camelContext>
  </beans>
```

NOTE In Camel 2.x we change to use `http://camel.apache.org/schema/cxf` as the CXF endpoint's target namespace.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://camel.apache.org/schema/cxf"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans-2.0.xsd
      http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd

       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
```

```
spring.xsd     ">
 ...
```

Note

In Fuse Mediation Router 2.x, the `http://activemq.apache.org/camel/schema/cxfEndpoint` namespace was changed to `http://camel.apache.org/schema/cxf`.

Be sure to include the JAX-WS `schemaLocation` attribute specified on the root beans element. This allows CXF to validate the file and is required. Also note the namespace declarations at the end of the `<cxf:cxfEndpoint/>` tag--these are required because the combined `{namespace}localName` syntax is presently not supported for this tag's attribute values.

The `cxf:cxfEndpoint` element supports many additional attributes:

Name	Value
`PortName`	The endpoint name this service is implementing, it maps to the `wsdl:port@name`. In the format of `ns:PORT_NAME` where `ns` is a namespace prefix valid at this scope.
`serviceName`	The service name this service is implementing, it maps to the `wsdl:service@name`. In the format of `ns:SERVICE_NAME` where `ns` is a namespace prefix valid at this scope.
`wsdlURL`	The location of the WSDL. Can be on the classpath, file system, or be hosted remotely.
`bindingId`	The `bindingId` for the service model to use.
`address`	The service publish address.
`bus`	The bus name that will be used in the JAX-WS endpoint.
`serviceClass`	The class name of the SEI (Service Endpoint Interface) class which could have JSR181 annotation or not.

It also supports many child elements:

Name	Value
`cxf:inInterceptors`	The incoming interceptors for this endpoint. A list of `<bean>` or `<ref>`.
`cxf:inFaultInterceptors`	The incoming fault interceptors for this endpoint. A list of `<bean>` or `<ref>`.
`cxf:outInterceptors`	The outgoing interceptors for this endpoint. A list of `<bean>` or `<ref>`.
`cxf:outFaultInterceptors`	The outgoing fault interceptors for this endpoint. A list of `<bean>` or `<ref>`.
`cxf:properties`	A properties map which should be supplied to the JAX-WS endpoint. See below.

`cxf:handlers`	A JAX-WS handler list which should be supplied to the JAX-WS endpoint. See below.
`cxf:dataBinding`	You can specify the which `DataBinding` will be use in the endpoint. This can be supplied using the Spring `<bean class="MyDataBinding"/>` syntax.
`cxf:binding`	You can specify the `BindingFactory` for this endpoint to use. This can be supplied using the Spring `<bean class="MyBindingFactory"/>` syntax.
`cxf:features`	The features that hold the interceptors for this endpoint. A list of `<bean>`s or `<ref>`s
`cxf:schemaLocations`	The schema locations for endpoint to use. A list of `<schemaLocation>`s
`cxf:serviceFactory`	The service factory for this endpoint to use. This can be supplied using the Spring `<bean class="MyServiceFactory"/>` syntax

You can find more advanced examples which show how to provide interceptors, properties and handlers here: http://cwiki.apache.org/CXF20DOC/jax-ws-configuration.html

Note

You can use CXF:properties to set the CXF endpoint's `dataFormat` and `setDefaultBus` properties from a Spring configuration file, as follows:

```
<cxf:cxfEndpoint id="testEndpoint" address="http://localhost:9000/router"
     serviceClass="org.apache.camel.component.cxf.HelloService"
     endpointName="s:PortName"
     serviceName="s:ServiceName"
     xmlns:s="http://www.example.com/test">
     <cxf:properties>
       <entry key="dataFormat" value="MESSAGE"/>
       <entry key="setDefaultBus" value="true"/>
     </cxf:properties>
   </cxf:cxfEndpoint>
```

How to make the camel-cxf component use log4j instead of java.util.logging

CXF's default logger is `java.util.logging`. If you want to change it to `log4j`, proceed as follows. Create a file, in the classpath, named `META-INF/cxf/org.apache.cxf.logger`. This file should contain the fully-qualified name of the class, `org.apache.cxf.common.logging.Log4jLogger`, with no comments, on a single line.

How to let camel-cxf response message with xml start document

If you are using some soap client such as PHP, you will get this kind of error, because CXF doesn't add the XML start document "<?xml version="1.0" encoding="utf-8"?>"

```
Error:sendSms: SoapFault exception: [Client] looks like we got no XML document in [...]
```

To resolved this issue, you just need to tell StaxOutInterceptor to write the XML start document for you.

```
public class WriteXmlDeclarationInterceptor extends AbstractPhaseInterceptor<SoapMessage>
{
    public WriteXmlDeclarationInterceptor() {
        super(Phase.PRE_STREAM);
        addBefore(StaxOutInterceptor.class.getName());
    }

    public void handleMessage(SoapMessage message) throws Fault {
        message.put("org.apache.cxf.stax.force-start-document", Boolean.TRUE);
    }

}
```

You can add a customer interceptor like this and configure it into you camel-cxf endpont

```
<cxf:cxfEndpoint id="routerEndpoint" address="http://localhost:9003/CamelContext/RouterPort"

   serviceClass="org.apache.hello_world_soap_http.GreeterImpl">
     <cxf:outInterceptors>
         <!-- This interceptor will force the CXF server send the XML start document to
client -->
         <bean class="org.apache.camel.component.cxf.WriteXmlDeclarationInterceptor"/>
     </cxf:outInterceptors>
     <cxf:properties>
         <!-- Set the publishedEndpointUrl which could override the service address from
generated WSDL as you want -->
         <entry key="publishedEndpointUrl" value="http://www.simple.com/services/test" />
     </cxf:properties>
 </cxf:cxfEndpoint>
```

Or adding a message header for it like this if you are using **Camel 2.4**.

```
 // set up the response context which force start document
 Map<String, Object> map = new HashMap<String, Object>();
 map.put("org.apache.cxf.stax.force-start-document", Boolean.TRUE);
 exchange.getOut().setHeader(Client.RESPONSE_CONTEXT, map);
```

How to consume a message from a camel-cxf endpoint in POJO data format

The `camel-cxf` endpoint consumer `POJO` data format is based on the cxf invoker[5], so the message header has a property with the name of `CxfConstants.OPERATION_NAME` and the message body is a list of the SEI method parameters.

```
public class PersonProcessor implements Processor {

    private static final transient Log LOG = LogFactory.getLog(PersonProcessor.class);

    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        LOG.info("processing exchange in camel");

        BindingOperationInfo boi = (BindingOperationInfo)exchange.getProperty(BindingOpera
tionInfo.class.toString());
        if (boi != null) {
            LOG.info("boi.isUnwrapped" + boi.isUnwrapped());
        }
        // Get the parameters list which element is the holder.
        MessageContentsList msgList = (MessageContentsList)exchange.getIn().getBody();
        Holder<String> personId = (Holder<String>)msgList.get(0);
        Holder<String> ssn = (Holder<String>)msgList.get(1);
        Holder<String> name = (Holder<String>)msgList.get(2);

        if (personId.value == null || personId.value.length() == 0) {
            LOG.info("person id 123, so throwing exception");
            // Try to throw out the soap fault message
            org.apache.camel.wsdl_first.types.UnknownPersonFault personFault =
                new org.apache.camel.wsdl_first.types.UnknownPersonFault();
            personFault.setPersonId("");
            org.apache.camel.wsdl_first.UnknownPersonFault fault =
                new org.apache.camel.wsdl_first.UnknownPersonFault("Get the null value of
person name", personFault);
            // Since camel has its own exception handler framework, we can't throw the ex
ception to trigger it
          // We just set the fault message in the exchange for camel-cxf component handling
 and return
            exchange.getOut().setFault(true);
            exchange.getOut().setBody(fault);
            return;
        }

        name.value = "Bonjour";
        ssn.value = "123";
```

[5] http://cwiki.apache.org/CXF20DOC/invokers.html

```
        LOG.info("setting Bonjour as the response");
        // Set the response message, first element is the return value of the operation,
        // the others are the holders of method parameters
        exchange.getOut().setBody(new Object[] {null, personId, ssn, name});
    }

}
```

How to prepare the message for the camel-cxf endpoint in POJO data format

The `camel-cxf` endpoint producer is based on the cxf client API[6]. First you need to specify the operation name in the message header, then add the method parameters to a list, and initialize the message with this parameter list. The response message's body is a `messageContentsList`, you can get the result from that list.

Note: After Fuse Mediation Router 1.5,the message body changed from an object array to a message content list. If you still want to get the object array from the message body, you can get the body using `message.getbody(Object[].class)`, as follows:

```
Exchange senderExchange = new DefaultExchange(context, ExchangePattern.InOut);
final List<String> params = new ArrayList<String>();
// Prepare the request message for the camel-cxf procedure
params.add(TEST_MESSAGE);
senderExchange.getIn().setBody(params);
senderExchange.getIn().setHeader(CxfConstants.OPERATION_NAME, ECHO_OPERATION);

Exchange exchange = template.send("direct:EndpointA", senderExchange);

org.apache.camel.Message out = exchange.getOut();
// The response message's body is an MessageContentsList which first element is the return
 value of the operation,
// If there are some holder parameters, the holder parameter will be filled in the reset
of List.
// The result will be extract from the MessageContentsList with the String class type
MessageContentsList result = (MessageContentsList)out.getBody();
LOG.info("Received output text: " + result.get(0));
Map<String, Object> responseContext = CastUtils.cast((Map)out.getHeader(Client.RESPONSE_CON
TEXT));
assertNotNull(responseContext);
assertEquals("We should get the response context here", "UTF-8", respon
seContext.get(org.apache.cxf.message.Message.ENCODING));
assertEquals("Reply body on Camel is wrong", "echo " + TEST_MESSAGE, result.get(0));
```

[6] https://svn.apache.org/repos/asf/cxf/trunk/api/src/main/java/org/apache/cxf/endpoint/Client.java

How to deal with the message for a camel-cxf endpoint in PAYLOAD data format

`PAYLOAD` means that you process the payload message from the SOAP envelope. You can use the `Header.HEADER_LIST` as the key to set or get the SOAP headers and use the `List<Element>` to set or get SOAP body elements. In Fuse Mediation Router 1.x, you can get the `List<Element>` and header from the CXF Message, but if you want to set the response message, you need to create the CXF message using the CXF API.

```
protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {
           from(SIMPLE_ENDPOINT_URI + "&dataFormat=PAYLOAD").to("log:info").process(new
Processor() {
               public void process(final Exchange exchange) throws Exception {
                   Message inMessage = exchange.getIn();
                   if (inMessage instanceof CxfMessage) {
                       CxfMessage cxfInMessage = (CxfMessage) inMessage;
                       CxfMessage cxfOutMessage = (CxfMessage) exchange.getOut();
                    List<Element> inElements = cxfInMessage.getMessage().get(List.class);

                       List<Element> outElements = new ArrayList<Element>();
                       XmlConverter converter = new XmlConverter();
                       String documentString = ECHO_RESPONSE;
                       if (inElements.get(0).getLocalName().equals("echoBoolean")) {
                           documentString = ECHO_BOOLEAN_RESPONSE;
                       }
                       org.apache.cxf.message.Exchange ex = ((CxfExchange)exchange).ge
tExchange();
                       Endpoint ep = ex.get(Endpoint.class);
                      org.apache.cxf.message.Message response = ep.getBinding().createMes
sage();
                       Document outDocument = converter.toDOMDocument(documentString);
                       outElements.add(outDocument.getDocumentElement());
                       response.put(List.class, outElements);
                       cxfOutMessage.setMessage(response);
                   }
               }
           });
       }
    };
}
```

In Fuse Mediation Router 2.0: `CxfMessage.getBody()` will return an `org.apache.camel.component.cxf.CxfPayload` object, which has getters for SOAP message headers and

Body elements. This change enables decoupling the native CXF message from the Fuse Mediation Router message.

```
protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {
            from(SIMPLE_ENDPOINT_URI + "&dataFormat=PAYLOAD").to("log:info").process(new
Processor() {
                @SuppressWarnings("unchecked")
                public void process(final Exchange exchange) throws Exception {

                   CxfPayload<SoapHeader> requestPayload = exchange.getIn().getBody(CxfPay
load.class);
                    List<Element> inElements = requestPayload.getBody();
                    List<Element> outElements = new ArrayList<Element>();
                  // You can use a customer toStringConverter to turn a CxfPayLoad message
 into String as you want
                    String request = exchange.getIn().getBody(String.class);
                    XmlConverter converter = new XmlConverter();
                    String documentString = ECHO_RESPONSE;
                    if (inElements.get(0).getLocalName().equals("echoBoolean")) {
                        documentString = ECHO_BOOLEAN_RESPONSE;
                       assertEquals("Get a wrong request", ECHO_BOOLEAN_REQUEST, request);

                    } else {
                        assertEquals("Get a wrong request", ECHO_REQUEST, request);
                    }
                    Document outDocument = converter.toDOMDocument(documentString);
                    outElements.add(outDocument.getDocumentElement());
                    // set the payload header with null
                 CxfPayload<SoapHeader> responsePayload = new CxfPayload<SoapHeader>(null,
 outElements);
                    exchange.getOut().setBody(responsePayload);
                }
            });
        }
    };
}
```

How to get and set SOAP headers in POJO mode

`POJO` means that the data format is a *list of Java objects* when the CXF endpoint produces or consumes Camel exchanges. Even though Fuse Mediation Router exposes the message body as POJOs in this mode, the CXF component still provides access to read and write SOAP headers. However, since CXF interceptors remove in-band SOAP headers from the header list after they have been processed, only out-of-band SOAP headers are available in POJO mode.

The following example illustrates how to get/set SOAP headers. Suppose we have a route that forwards from one CXF endpoint to another. That is, SOAP Client -> Fuse Mediation Router -> CXF service. We can attach two processors to obtain/insert SOAP headers at (1) before request goes out to the CXF service and (2) before response comes back to the SOAP Client. Processor (1) and (2) in this example are InsertRequestOutHeaderProcessor and InsertResponseOutHeaderProcessor. Our route looks like this:

```
<route>
    <from uri="cxf:bean:routerRelayEndpointWithInsertion"/>
    <process ref="InsertRequestOutHeaderProcessor" />
    <to uri="cxf:bean:serviceRelayEndpointWithInsertion"/>
    <process ref="InsertResponseOutHeaderProcessor" />
</route>
```

In 2.x SOAP headers are propagated to and from Fuse Mediation Router Message headers. The Fuse Mediation Router message header name is `org.apache.cxf.headers.Header.list`, which is a constant defined in CXF (`org.apache.cxf.headers.Header.HEADER_LIST`). The header value is a `List<>` of CXF `SoapHeader` objects (`org.apache.cxf.binding.soap.SoapHeader`). The following snippet is the `InsertResponseOutHeaderProcessor` (that inserts a new SOAP header in the response message). The way to access SOAP headers in both `InsertResponseOutHeaderProcessor` and `InsertRequestOutHeaderProcessor` are actually the same. The only difference between the two processors is setting the direction of the inserted SOAP header.

```
public static class InsertResponseOutHeaderProcessor implements Processor {

    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
       List<SoapHeader> soapHeaders = (List)exchange.getIn().getHeader(Header.HEADER_LIST);


        // Insert a new header
        String xml = "<?xml version=\"1.0\" encoding=\"utf-8\"?><outofbandHeader "
            + "xmlns=\"http://cxf.apache.org/outofband/Header\" hdrAttribute=\"testHdrAt
tribute\" "
            + "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\" soap:mustUnder
stand=\"1\">"
            + "<name>New_testOobHeader</name><value>New_testOobHeaderValue</value></outof
bandHeader>";
        SoapHeader newHeader = new SoapHeader(soapHeaders.get(0).getName(),
                       DOMUtils.readXml(new StringReader(xml)).getDocumentElement());
        // make sure direction is OUT since it is a response message.
        newHeader.setDirection(Direction.DIRECTION_OUT);
        //newHeader.setMustUnderstand(false);
        soapHeaders.add(newHeader);

    }

}
```

In 1.x SOAP headers are not propagated to and from Fuse Mediation Router Message headers. Users have to go deeper into CXF APIs to access SOAP headers. Also, accessing the SOAP headers in a request message is slightly different than in a response message. The `InsertRequestOutHeaderProcessor` and `InsertResponseOutHeaderProcessor` are as follow.s:

```
public static class InsertRequestOutHeaderProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        CxfMessage message = exchange.getIn().getBody(CxfMessage.class);
        Message cxf = message.getMessage();
        List<SoapHeader> soapHeaders = (List)cxf.get(Header.HEADER_LIST);

        // Insert a new header
        String xml = "<?xml version=\"1.0\" encoding=\"utf-8\"?><outofbandHeader "
            + "xmlns=\"http://cxf.apache.org/outofband/Header\" hdrAttribute=\"testHdrAt
tribute\" "
            + "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\" soap:mustUnder
stand=\"1\">"
            + "<name>New_testOobHeader</name><value>New_testOobHeaderValue</value></outof
bandHeader>";

        SoapHeader newHeader = new SoapHeader(soapHeaders.get(0).getName(),
                                              DOMUtils.readXml(new StringReader(xml)).get
DocumentElement());
        // make sure direction is IN since it is a request message.
        newHeader.setDirection(Direction.DIRECTION_IN);
        //newHeader.setMustUnderstand(false);
        soapHeaders.add(newHeader);

    }
}

public static class InsertResponseOutHeaderProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        CxfMessage message = exchange.getIn().getBody(CxfMessage.class);
        Map responseContext = (Map)message.getMessage().get(Client.RESPONSE_CONTEXT);
        List<SoapHeader> soapHeaders = (List)responseContext.get(Header.HEADER_LIST);

        // Insert a new header
        String xml = "<?xml version=\"1.0\" encoding=\"utf-8\"?><outofbandHeader "
            + "xmlns=\"http://cxf.apache.org/outofband/Header\" hdrAttribute=\"testHdrAt
tribute\" "
            + "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\" soap:mustUnder
stand=\"1\">"
            + "<name>New_testOobHeader</name><value>New_testOobHeaderValue</value></outof
bandHeader>";
        SoapHeader newHeader = new SoapHeader(soapHeaders.get(0).getName(),
                        DOMUtils.readXml(new StringReader(xml)).getDocumentElement());
        // make sure direction is OUT since it is a response message.
        newHeader.setDirection(Direction.DIRECTION_OUT);
```

```
        //newHeader.setMustUnderstand(false);
        soapHeaders.add(newHeader);

    }
}
```

How to get and set SOAP headers in PAYLOAD mode

We have already shown how to access SOAP message (`CxfPayload` object) in `PAYLOAD` mode (see "How to deal with the message for a camel-cxf endpoint in PAYLOAD data format" on page 88).

In Fuse Mediation Router 2.x Once you obtain a `CxfPayload` object, you can invoke the `CxfPayload.getHeaders()` method that returns a `List` of DOM Elements (SOAP headers).

```
from(getRouterEndpointURI()).process(new Processor() {
    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        CxfPayload<SoapHeader> payload = exchange.getIn().getBody(CxfPayload.class);
        List<Element> elements = payload.getBody();
        assertNotNull("We should get the elements here", elements);
        assertEquals("Get the wrong elements size", 1, elements.size());
        assertEquals("Get the wrong namespace URI", "http://camel.apache.org/pizza/types",

                elements.get(0).getNamespaceURI());

        List<SoapHeader> headers = payload.getHeaders();
        assertNotNull("We should get the headers here", headers);
        assertEquals("Get the wrong headers size", headers.size(), 1);
        assertEquals("Get the wrong namespace URI",
                ((Element)(headers.get(0).getObject())).getNamespaceURI(),
                "http://camel.apache.org/pizza/types");
    }

})
.to(getServiceEndpointURI());
```

In Fuse Mediation Router 1.x: You can get/set to the CXF Message by the key `org.apache.cxf.headers.Header.list` which is a constant defined in CXF (`org.apache.cxf.headers.Header.HEADER_LIST`).

```
from(routerEndpointURI).process(new Processor() {
    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        Message inMessage = exchange.getIn();
        CxfMessage message = (CxfMessage) inMessage;
        List<Element> elements = message.getMessage().get(List.class);
        assertNotNull("We should get the payload elements here" , elements);
```

```
        assertEquals("Get the wrong elements size" , elements.size(), 1);
        assertEquals("Get the wrong namespace URI" , elements.get(0).getNamespaceURI(),
"http://camel.apache.org/pizza/types");

        List<SoapHeader> headers = CastUtils.cast((List<?>)message.getMessage().get(Head
er.HEADER_LIST));
        assertNotNull("We should get the headers here", headers);
        assertEquals("Get the wrong headers size", headers.size(), 1);
        assertEquals("Get the wrong namespace URI" , ((Element)(headers.get(0).getOb
ject())).getNamespaceURI(), "http://camel.apache.org/pizza/types");

    }

})
.to(serviceEndpointURI);
```

SOAP headers are not available in MESSAGE mode

SOAP headers are not available in `MESSAGE` mode as SOAP processing is skipped.

How to throw a SOAP Fault from Fuse Mediation Router

If you are using a CXF endpoint to consume the SOAP request, you may need to throw the SOAP `Fault` from the camel context. Basically, you can use the `throwFault` DSL to do that; it works for `POJO`, `PAYLOAD` and `MESSAGE` data format. You can define the soap fault like this:

```
SOAP_FAULT = new SoapFault(EXCEPTION_MESSAGE, SoapFault.FAULT_CODE_CLIENT);
Element detail = SOAP_FAULT.getOrCreateDetail();
Document doc = detail.getOwnerDocument();
Text tn = doc.createTextNode(DETAIL_TEXT);
detail.appendChild(tn);
```

Then throw it as you like:

```
from(routerEndpointURI).setFaultBody(constant(SOAP_FAULT));
```

If your CXF endpoint is working in the `MESSAGE` data format, you could set the the SOAP Fault message in the message body and set the response code in the message header.

```
from(routerEndpointURI).process(new Processor() {

    public void process(Exchange exchange) throws Exception {
        Message out = exchange.getOut();
        // Set the message body with the
        out.setBody(this.getClass().getResourceAsStream("SoapFaultMessage.xml"));
        // Set the response code here
        out.setHeader(org.apache.cxf.message.Message.RESPONSE_CODE, new Integer(500));
```

```
    }

});
```

Note

The response code setting only works in Fuse Mediation Router version >= 1.5.1

How to propagate a CXF endpoint's request and response context

cxf client API[7] provides a way to invoke the operation with request and response context. If you are using a CXF endpoint producer to invoke the external Web service, you can set the request context and get the response context with the following code:

```
       CxfExchange exchange = (CxfExchange)template.send(getJaxwsEndpointUri(), new Pro
cessor() {
            public void process(final Exchange exchange) {
                final List<String> params = new ArrayList<String>();
                params.add(TEST_MESSAGE);
                // Set the request context to the inMessage
                Map<String, Object> requestContext = new HashMap<String, Object>();
                requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, JAXWS_SERV
ER_ADDRESS);
                exchange.getIn().setBody(params);
                exchange.getIn().setHeader(Client.REQUEST_CONTEXT , requestContext);
              exchange.getIn().setHeader(CxfConstants.OPERATION_NAME, GREET_ME_OPERATION);

            }
        });
        org.apache.camel.Message out = exchange.getOut();
        // The output is an object array, the first element of the array is the return
value
        Object\[\] output = out.getBody(Object\[\].class);
        LOG.info("Received output text: " + output\[0\]);
        // Get the response context form outMessage
        Map<String, Object> responseContext = CastUtils.cast((Map)out.getHeader(Client.RE
SPONSE_CONTEXT));
        assertNotNull(responseContext);
        assertEquals("Get the wrong wsdl opertion name", "{ht
tp://apache.org/hello_world_soap_http}greetMe",
                     responseContext.get("javax.xml.ws.wsdl.operation").toString());
```

[7] https://svn.apache.org/repos/asf/cxf/trunk/api/src/main/java/org/apache/cxf/endpoint/Client.java

Attachment Support

POJO Mode: Both SOAP with Attachment and MTOM are supported (see example in Payload Mode for enabling MTOM).However, SOAP with Attachment is not tested.Since attachments are marshalled and unmarshalled into POJOs, users typically do not need to deal with the attachment themself.Attachments are propagated to Camel message's attachments since 2.1.So, it is possible to retreive attachments by Camel Message API

```
DataHandler Message.getAttachment(String id)
```

.

Payload Mode: MTOM is supported since 2.1. Attachments can be retrieved by Camel Message APIs mentioned above. SOAP with Attachment is not supported as there is no SOAP processing in this mode.

To enable MTOM, set the CXF endpoint property "mtom_enabled" to *true*. (I believe you can only do it with Spring.)

```
<cxf:cxfEndpoint id="routerEndpoint" address="http://localhost:9091/jaxws-mtom/hello"
         wsdlURL="mtom.wsdl"
         serviceName="ns:HelloService"
         endpointName="ns:HelloPort"
         xmlns:ns="http://apache.org/camel/cxf/mtom_feature">

     <cxf:properties>
         <!--  enable mtom by setting this property to true -->
         <entry key="mtom-enabled" value="true"/>

         <!--  set the camel-cxf endpoint data fromat to PAYLOAD mode -->
         <entry key="dataFormat" value="PAYLOAD"/>
     </cxf:properties>
```

You can produce a Camel message with attachment to send to a CXF endpoint in Payload mode.

```
Exchange exchange = context.createProducerTemplate().send("direct:testEndpoint", new Pro
cessor() {

    public void process(Exchange exchange) throws Exception {
        exchange.setPattern(ExchangePattern.InOut);
        List<Element> elements = new ArrayList<Element>();
        elements.add(DOMUtils.readXml(new StringReader(MtomTestHelper.REQ_MESSAGE)).getDoc
umentElement());
       CxfPayload<SoapHeader> body = new CxfPayload<SoapHeader>(new ArrayList<SoapHeader>(),
```

```
            elements);
        exchange.getIn().setBody(body);
        exchange.getIn().addAttachment(MtomTestHelper.REQ_PHOTO_CID,
            new DataHandler(new ByteArrayDataSource(MtomTestHelper.REQ_PHOTO_DATA, "applic
ation/octet-stream")));

        exchange.getIn().addAttachment(MtomTestHelper.REQ_IMAGE_CID,
            new DataHandler(new ByteArrayDataSource(MtomTestHelper.requestJpeg, "im
age/jpeg")));

    }

});

// process response

CxfPayload<SoapHeader> out = exchange.getOut().getBody(CxfPayload.class);
Assert.assertEquals(1, out.getBody().size());

Map<String, String> ns = new HashMap<String, String>();
ns.put("ns", MtomTestHelper.SERVICE_TYPES_NS);
ns.put("xop", MtomTestHelper.XOP_NS);

XPathUtils xu = new XPathUtils(ns);
Element ele = (Element)xu.getValue("//ns:DetailResponse/ns:photo/xop:Include", out.get
Body().get(0),
                                   XPathConstants.NODE);
String photoId = ele.getAttribute("href").substring(4); // skip "cid:"

ele = (Element)xu.getValue("//ns:DetailResponse/ns:image/xop:Include", out.getBody().get(0),

                                   XPathConstants.NODE);
String imageId = ele.getAttribute("href").substring(4); // skip "cid:"

DataHandler dr = exchange.getOut().getAttachment(photoId);
Assert.assertEquals("application/octet-stream", dr.getContentType());
MtomTestHelper.assertEquals(MtomTestHelper.RESP_PHOTO_DATA, IOUtils.readBytesFromStream(dr.get
InputStream()));

dr = exchange.getOut().getAttachment(imageId);
Assert.assertEquals("image/jpeg", dr.getContentType());

BufferedImage image = ImageIO.read(dr.getInputStream());
Assert.assertEquals(560, image.getWidth());
Assert.assertEquals(300, image.getHeight());
```

You can also consume a Camel message received from a CXF endpoint in Payload mode.

```
public static class MyProcessor implements Processor {

    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        CxfPayload<SoapHeader> in = exchange.getIn().getBody(CxfPayload.class);

        // verify request
        Assert.assertEquals(1, in.getBody().size());

        Map<String, String> ns = new HashMap<String, String>();
        ns.put("ns", MtomTestHelper.SERVICE_TYPES_NS);
        ns.put("xop", MtomTestHelper.XOP_NS);

        XPathUtils xu = new XPathUtils(ns);
        Element ele = (Element)xu.getValue("//ns:Detail/ns:photo/xop:Include", in.get
Body().get(0),
                                           XPathConstants.NODE);
        String photoId = ele.getAttribute("href").substring(4); // skip "cid:"
        Assert.assertEquals(MtomTestHelper.REQ_PHOTO_CID, photoId);

       ele = (Element)xu.getValue("//ns:Detail/ns:image/xop:Include", in.getBody().get(0),

                                           XPathConstants.NODE);
        String imageId = ele.getAttribute("href").substring(4); // skip "cid:"
        Assert.assertEquals(MtomTestHelper.REQ_IMAGE_CID, imageId);

        DataHandler dr = exchange.getIn().getAttachment(photoId);
        Assert.assertEquals("application/octet-stream", dr.getContentType());
        MtomTestHelper.assertEquals(MtomTestHelper.REQ_PHOTO_DATA, IOUtils.readBytesFrom
Stream(dr.getInputStream()));

        dr = exchange.getIn().getAttachment(imageId);
        Assert.assertEquals("image/jpeg", dr.getContentType());
        MtomTestHelper.assertEquals(MtomTestHelper.requestJpeg, IOUtils.readBytesFrom
Stream(dr.getInputStream()));

        // create response
        List<Element> elements = new ArrayList<Element>();
       elements.add(DOMUtils.readXml(new StringReader(MtomTestHelper.RESP_MESSAGE)).getDoc
umentElement());
       CxfPayload<SoapHeader> body = new CxfPayload<SoapHeader>(new ArrayList<SoapHeader>(),

            elements);
        exchange.getOut().setBody(body);
        exchange.getOut().addAttachment(MtomTestHelper.RESP_PHOTO_CID,
           new DataHandler(new ByteArrayDataSource(MtomTestHelper.RESP_PHOTO_DATA, "applic
ation/octet-stream")));

        exchange.getOut().addAttachment(MtomTestHelper.RESP_IMAGE_CID,
```

```
            new DataHandler(new ByteArrayDataSource(MtomTestHelper.responseJpeg, "im
age/jpeg")));

    }
}
```

Message Mode: Attachments are not supported as it does not process the message at all.

Chapter 15. CXFRS

CXFRS Component

The **cxfrs:** component provides integration with Apache CXF[1] for connecting to JAX-RS services hosted in CXF.

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

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

Note

When using CXF as a consumer, the CAMEL:CXF Bean Component allows you to factor out how message payloads are received from their processing as a RESTful or SOAP web service. This has the potential of using a multitude of transports to consume web services. The bean component's configuration is also simpler and provides the fastest method to implement web services using Camel and CXF.

URI format

```
cxfrs://address?options
```

Where **address** represents the CXF endpoint's address

```
cxfrs:bean:rsEndpoint
```

Where **rsEndpoint** represents the Spring bean's name which represents the CXFRS client or server

For either style above, you can append options to the URI as follows:

```
cxfrs:bean:cxfEndpoint?resourceClasses=org.apache.camel.rs.Example
```

[1] http://incubator.apache.org/cxf/

Options

Name	Description	Example
`resourceClasses`	The resource classes which you want to export as REST service	resourceClasses=org.apache.camel.rs.Example1,org.apac
`httpClientAPI`	**New to Fuse Mediation Router 2.1** If it is true, the CxfRsProducer will use the HttpClientAPI to invoke the service	`httpClientAPI=true`
`synchronous`	New in 2.5, this option will let CxfRsConsumer decide to use sync or async API to do the underlying work. The default value is false which means it will try to use async API by default.	`synchronous=true`
`throwExceptionOnFailure`	New in 2.6, this option tells the CxfRsProducer to inspect return codes and will generate an Exception if the return code is larger than 207.	`throwExceptionOnFailure=true`
`maxClientCacheSize`	New in 2.6, you can set the *In* message header, `CamelDestinationOverrideUrl`, to dynamically override the target destination Web Service or REST Service defined in your routes. The implementation caches CXF clients or `ClientFactoryBean` in `CxfProvider` and `CxfRsProvider`. This option allows you to configure the maximum size of the cache.	`maxClientCacheSize=5`

You can also configure the CXF REST endpoint through the Spring configuration. Since there are lots of difference between the CXF REST client and CXF REST Server, we provide different configuration for them. Please check out the schema file[2] and CXF REST user guide[3] for more information.

[2] http://svn.apache.org/repos/asf/camel/trunk/components/camel-cxf/src/main/resources/schema/cxfEndpoint.xsd
[3] http://cwiki.apache.org/CXF20DOC/jax-rs.html

How to configure the REST endpoint in Fuse Mediation Router

In camel-cxf schema file[4], there are two elements for the REST endpoint definition. **cxf:rsServer** for REST consumer, **cxf:rsClient** for REST producer. You can find a Fuse Mediation Router REST service route configuration example here.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://camel.apache.org/schema/cxf"
       xmlns:jaxrs="http://cxf.apache.org/jaxrs"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans.xsd
      http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd

       http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd
    ">
  <!-- Defined the real JAXRS back end service  -->
  <jaxrs:server id="restService"
          address="http://localhost:9002/rest"
          staticSubresourceResolution="true">
    <jaxrs:serviceBeans>
      <ref bean="customerService"/>
    </jaxrs:serviceBeans>
  </jaxrs:server>

  <bean id="jsonProvider" class="org.apache.cxf.jaxrs.provider.JSONProvider"/>

  <bean id="customerService" class="org.apache.camel.component.cxf.jaxrs.testbean.Custom
erService" />

  <!-- Defined the server endpoint to create the cxf-rs consumer -->
  <cxf:rsServer id="rsServer" address="http://localhost:9000/route"
    serviceClass="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService" />

  <!-- Defined the client endpoint to create the cxf-rs consumer -->
  <cxf:rsClient id="rsClient" address="http://localhost:9002/rest"
    serviceClass="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService"/>

  <!-- The camel route context -->
  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
       <from uri="cxfrs://bean://rsServer"/>
```

[4] http://svn.apache.org/repos/asf/camel/trunk/components/camel-cxf/src/main/resources/schema/cxfEndpoint.xsd

```
      <!-- We can remove this configure as the CXFRS producer is using the HttpAPI by default
 -->
       <setHeader headerName="CamelCxfRsUsingHttpAPI">
         <constant>True</constant>
       </setHeader>
       <to uri="cxfrs://bean://rsClient"/>
    </route>
  </camelContext>

</beans>
```

How to consume the REST request in Fuse Mediation Router

CXF JAXRS front end[5] implements the JAXRS(JSR311) API[6], so we can export the resources classes as a REST service. And we leverage the CXF Invoker API[7] to turn a REST request into a normal Java object method invocation. Unlike the `camel-restlet`, you don't need to specify the URI template within your restlet endpoint, CXF take care of the REST request URI to resource class method mapping according to the JSR311 specification. All you need to do in Fuse Mediation Router is delegate this method request to a right processor or endpoint.

Here is an example

```
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() {
            errorHandler(new NoErrorHandlerBuilder());
            from(CXF_RS_ENDPOINT_URI).process(new Processor() {

                public void process(Exchange exchange) throws Exception {
                    Message inMessage = exchange.getIn();
                    // Get the operation name from in message
                   String operationName = inMessage.getHeader(CxfConstants.OPERATION_NAME,
 String.class);
                    if ("getCustomer".equals(operationName)) {
                        String httpMethod = inMessage.getHeader(Exchange.HTTP_METHOD,
String.class);
                        assertEquals("Get a wrong http method", "GET", httpMethod);
                      String path = inMessage.getHeader(Exchange.HTTP_PATH, String.class);

                        // The parameter of the invocation is stored in the body of in
message
                        String id = (String) inMessage.getBody(String.class);
                        if ("/customerservice/customers/126".equals(path)) {
```

[5] http://cwiki.apache.org/CXF20DOC/jax-rs.html
[6] https://jsr311.dev.java.net/
[7] http://cwiki.apache.org/confluence/display/CXF20DOC/Invokers

```
                            Customer customer = new Customer();
                            customer.setId(Long.parseLong(id));
                            customer.setName("Willem");
                            // We just put the response Object into the out message body
                            exchange.getOut().setBody(customer);
                        } else {
                            if ("/customerservice/customers/456".equals(path)) {
                                Response r = Response.status(404).entity("Can't found the
customer with uri " + path).build();
                                throw new WebApplicationException(r);
                            } else {
                                throw new RuntimeCamelException("Can't found the customer
with uri " + path);
                            }
                        }
                    }
                    if ("updateCustomer".equals(operationName)) {
                     assertEquals("Get a wrong customer message header", "header1;header2",
 inMessage.getHeader("test"));
                        String httpMethod = inMessage.getHeader(Exchange.HTTP_METHOD,
String.class);
                        assertEquals("Get a wrong http method", "PUT", httpMethod);
                        Customer customer = inMessage.getBody(Customer.class);
                        assertNotNull("The customer should not be null.", customer);
                        // Now you can do what you want on the customer object
                        assertEquals("Get a wrong customer name.", "Mary", customer.get
Name());
                        // set the response back
                        exchange.getOut().setBody(Response.ok().build());
                }
                }

            });
        }
    };
}
```

How to invoke the REST service through camel-cxfrs producer ?

CXF JAXRS front end[8] implements a proxy based client API[9], with this API you can invoke the remote REST service through a proxy. `camel-cxfrs` producer is based on this proxy API[10]. So, you just need to specify the operation name in the message header and prepare the parameter in the message body, `camel-cxfrs` producer will generate right REST request for you.

[8] http://cwiki.apache.org/CXF20DOC/jax-rs.html
[9] http://cwiki.apache.org/CXF20DOC/jax-rs.html#JAX-RS-ProxybasedAPI
[10] http://cwiki.apache.org/CXF20DOC/jax-rs.html#JAX-RS-ProxybasedAPI

Here is an example

```
Exchange exchange = template.send("direct://proxy", new Processor() {

    public void process(Exchange exchange) throws Exception {
        exchange.setPattern(ExchangePattern.InOut);
        Message inMessage = exchange.getIn();
        // set the operation name
        inMessage.setHeader(CxfConstants.OPERATION_NAME, "getCustomer");
        // using the proxy client API
        inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_USING_HTTP_API, Boolean.FALSE);
        // set the parameters , if you just have one parameter
        // camel will put this object into an Object[] itself
        inMessage.setBody("123");
    }

});

// get the response message
Customer response = (Customer) exchange.getOut().getBody();

assertNotNull("The response should not be null ", response);
assertEquals("Get a wrong customer id ", String.valueOf(response.getId()), "123");
assertEquals("Get a wrong customer name", response.getName(), "John");
```

CXF JAXRS front end[11] also provides a http centric client API[12], You can also invoke this API from `camel-cxfrs` producer. You need to specify the HTTP_PATH and Http method and let the the producer know to use the HTTP centric client by using the URI option **httpClientAPI** or set the message header with `CxfConstants.CAMEL_CXF_RS_USING_HTTP_API`. You can turn the response object to the type class that you specify with `CxfConstants.CAMEL_CXF_RS_RESPONSE_CLASS`.

```
Exchange exchange = template.send("direct://http", new Processor() {

    public void process(Exchange exchange) throws Exception {
        exchange.setPattern(ExchangePattern.InOut);
        Message inMessage = exchange.getIn();
        // using the http central client API
        inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_USING_HTTP_API, Boolean.TRUE);
        // set the Http method
        inMessage.setHeader(Exchange.HTTP_METHOD, "GET");
        // set the relative path
        inMessage.setHeader(Exchange.HTTP_PATH, "/customerservice/customers/123");

        // Specify the response class , cxfrs will use InputStream as the response object
type
        inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_RESPONSE_CLASS, Customer.class);
```

[11] http://cwiki.apache.org/CXF20DOC/jax-rs.html
[12] http://cxf.apache.org/docs/jax-rs.html#JAX-RS-HTTPcentricclients

```
        // since we use the Get method, so we don't need to set the message body
        inMessage.setBody(null);
    }

});

// get the response message
Customer response = (Customer) exchange.getOut().getBody();

assertNotNull("The response should not be null ", response);
assertEquals("Get a wrong customer id ", String.valueOf(response.getId()), "123");
assertEquals("Get a wrong customer name", response.getName(), "John");
```

From Fuse Mediation Router 2.1, we also support to specify the query parameters from CXFRS URI for the CXFRS HTTP centric client.

```
Exchange exchange = template.send("cxfrs://http://localhost:9003/testQuery?httpCli
entAPI=true&q1=12&q2=13"
```

To support the Dynamical routing, you can override the URI's query parameters by using the `CxfConstants.CAMEL_CXF_RS_QUERY_MAP` header to set the parameter map for it.To support the Dynamical routing, you can override the URI's query parameters by using the `CxfConstants.CAMEL_CXF_RS_QUERY_MAP` header to set the parameter map for it.

```
Map<String, String> queryMap = new LinkedHashMap<String, String>();
queryMap.put("q1", "new");
queryMap.put("q2", "world");
inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_QUERY_MAP, queryMap);
```

Chapter 16. DataSet

DataSet Component

The DataSet component (available since 1.3.0) provides a mechanism to easily perform load & soak testing of your system. It works by allowing you to create DataSet instances[1] both as a source of messages and as a way to assert that the data set is received.

Fuse Mediation Router will use the throughput logger on page 331 when sending dataset's.

URI format

```
dataset:name[?options]
```

Where **name** is used to find the DataSet instance[2] in the Registry

Fuse Mediation Router ships with a support implementation of `org.apache.camel.component.dataset.DataSet`, the `org.apache.camel.component.dataset.DataSetSupport` class, that can be used as a base for implementing your own DataSet. Fuse Mediation Router also ships with a default implementation, the `org.apache.camel.component.dataset.SimpleDataSet` that can be used for testing.

Options

Option	Default	Description
`produceDelay`	3	Allows a delay in ms to be specified, which causes producers to pause in order to simulate slow producers. Uses a minimum of 3 ms delay unless you set this option to -1 to force no delay at all.
`consumeDelay`	0	Allows a delay in ms to be specified, which causes consumers to pause in order to simulate slow consumers.
`preloadSize`	0	Sets how many messages should be preloaded (sent) before the route completes its initialization.
`initialDelay`	1000	**Camel 2.1:** Time period in millis to wait before starting sending messages.
`minRate`	0	Wait until the DataSet contains at least this number of messages

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

[1] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/dataset/DataSet.html
[2] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/dataset/DataSet.html

Configuring DataSet

Fuse Mediation Router will lookup in the Registry for a bean implementing the DataSet interface. So you can register your own DataSet as:

```
<bean id="myDataSet" class="com.mycompany.MyDataSet">
   <property name="size" value="100"/>
</bean>
```

Example

For example, to test that a set of messages are sent to a queue and then consumed from the queue without losing any messages:

```
// send the dataset to a queue
from("dataset:foo").to("activemq:SomeQueue");

// now lets test that the messages are consumed correctly
from("activemq:SomeQueue").to("dataset:foo");
```

The above would look in the Registry to find the **foo** DataSet instance which is used to create the messages.

Then you create a DataSet implementation, such as using the `SimpleDataSet` as described below, configuring things like how big the data set is and what the messages look like etc.

Properties on SimpleDataSet

Property	Type	Description
`defaultBody`	`Object`	Specifies the default message body. For `SimpleDataSet` it is a constant payload; though if you want to create custom payloads per message, create your own derivation of `DataSetSupport`.
`reportGroup`	`long`	Specifies the number of messages to be received before reporting progress. Useful for showing progress of a large load test.
`size`	`long`	Specifies how many messages to send/consume.

- Spring Testing

Chapter 17. Db4o

Db4o Component

Available as of Camel 2.5

The **db4o:** component allows you to work with db4o[1] NoSQL database. The camel-db4o library is provided by the Camel Extra[2] project which hosts all *GPL related components for Camel.

Sending to the endpoint

Sending POJO object to the db4o endpoint adds and saves object into the database. The body of the message is assumed to be a POJO that has to be saved into the db40 database store.

Consuming from the endpoint

Consuming messages removes (or updates) POJO objects in the database. This allows you to use a Db4o datastore as a logical queue; consumers take messages from the queue and then delete them to logically remove them from the queue.

If you do not wish to delete the object when it has been processed, you can specify `consumeDelete=false` on the URI. This will result in the POJO being processed each poll.

URI format

```
db4o:className[?options]
```

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

Options

Name	Default Value	Description
`consumeDelete`	`true`	Option for `Db4oConsumer` only. Specifies whether or not the entity is deleted after it is consumed.
`consumer.delay`	`500`	Option for HibernateConsumer only. Delay in millis between each poll.
`consumer.initialDelay`	`1000`	Option for HibernateConsumer only. Millis before polling starts.

[1] http://www.db4o.com
[2] http://code.google.com/p/camel-extra/

`consumer.userFixedDelay false`	Option for HibernateConsumer only. Set to `true` to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService[3] in JDK for details.

[3] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html

Chapter 18. Direct

Direct Component

The **direct:** component provides direct, synchronous invocation of any consumers when a producer sends a message exchange. This endpoint can be used to connect existing routes in the **same** camel context.

Asynchronous

The Seda component provides asynchronous invocation of any consumers when a producer sends a message exchange.

Connection to other camel contexts

The VM on page 519 component provides connections between Camel contexts as long they run in the same **JVM**.

URI format

```
direct:someName[?options]
```

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

Options

Name	Default Value	Description
`allowMultipleConsumers`	`true`	**@deprecated** If set to `false`, then when a second consumer is started on the endpoint, an `IllegalStateException` is thrown. **Will be removed in Camel 2.1:** Direct endpoint does not support multiple consumers.

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

Samples

In the route below we use the direct component to link the two routes together:

```
from("activemq:queue:order.in").to("bean:orderServer?method=validate").to("direct:pro
cessOrder");

from("direct:processOrder").to("bean:orderService?method=process").to("activemq:queue:or
der.out");
```

And the sample using spring DSL:

```
   <route>
     <from uri="activemq:queue:order.in"/>
     <to uri="bean:orderService?method=validate"/>
     <to uri="direct:processOrder"/>
  </route>

  <route>
     <from uri="direct:processOrder"/>
     <to uri="bean:orderService?method=process"/>
     <to uri="activemq:queue:order.out"/>
  </route>
```

See also samples from the Seda component, how they can be used together.

- Seda
- VM on page 519

Chapter 19. EJB

EJB Component

Available as of 2.4

The **ejb:** component binds EJBs to message exchanges.

URI format

```
ejb:ejbName[?options]
```

Where **ejbName** can be any string which is used to look up the EJB in the Application Server JNDI Registry

Options

Name	Type	Default	Description
`method`	`String`	`null`	The method name that bean will be invoked. If not provided, will try to pick the method itself. In case of ambiguity an exception is thrown. See Bean Binding for more details.
`multiParameterArray`	`boolean`	`false`	How to treat the parameters which are passed from the message body; if it is `true`, the In message body should be an array of parameters.

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

The EJB on page 113 component extends the Bean on page 41 component in which most of the details from the Bean on page 41 component applies to this component as well.

Bean Binding

How bean methods to be invoked are chosen (if they are not specified explicitly through the **method** parameter) and how parameter values are constructed from the MessageMessage are all defined by the Bean Binding mechanism which is used throughout all of the various Bean Integration mechanisms in .

Examples

In the following examples we use the Greater EJB which is defined as follows:

```
public interface GreaterLocal {

    String hello(String name);

    String bye(String name);

}
```

And the implementation

```
@Stateless
public class GreaterImpl implements GreaterLocal {

    public String hello(String name) {
        return "Hello " + name;
    }

    public String bye(String name) {
        return "Bye " + name;
    }

}
```

Using Java DSL

In this example we want to invoke the `hello` method on the EJB. Since this example is based on an unit test using Apache OpenEJB we have to set a `JndiContext` on the EJB on page 113 component with the OpenEJB settings.

```
@Override
protected CamelContext createCamelContext() throws Exception {
    CamelContext answer = new DefaultCamelContext();

    // enlist EJB component using the JndiContext
    EjbComponent ejb = answer.getComponent("ejb", EjbComponent.class);
    ejb.setContext(createEjbContext());

    return answer;
}

private static Context createEjbContext() throws NamingException {
    // here we need to define our context factory to use OpenEJB for our testing
    Properties properties = new Properties();
    properties.setProperty(Context.INITIAL_CONTEXT_FACTORY, "org.apache.openejb.client.Loc
alInitialContextFactory");

    return new InitialContext(properties);
}
```

Then we are ready to use the EJB in the route:

```
from("direct:start")
    // invoke the greeter EJB using the local interface and invoke the hello method
    .to("ejb:GreaterImplLocal?method=hello")
    .to("mock:result");
```

In a real application server

In a real application server you most likely do not have to setup a `JndiContext` on the EJB on page 113 component as it will create a default `JndiContext` on the same JVM as the application server, which usually allows it to access the JNDI registry and lookup the EJB on page 113s. However if you need to access a application server on a remote JVM or the likes, you have to prepare the properties beforehand.

Using Spring XML

And this is the same example using Spring XML instead:

Again since this is based on an unit test we need to setup the EJB on page 113 component:

```
<!-- setup Camel EJB component -->
<bean id="ejb" class="org.apache.camel.component.ejb.EjbComponent">
    <property name="properties" ref="jndiProperties"/>
</bean>

<!-- use OpenEJB context factory -->
<p:properties id="jndiProperties">
    <prop key="java.naming.factory.initial">org.apache.openejb.client.LocalInitialContext
Factory</prop>
</p:properties>
```

Before we are ready to use EJB on page 113 in the routes:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <to uri="ejb:GreaterImplLocal?method=hello"/>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

- Bean on page 41
- Bean Binding

- Bean Integration

Chapter 20. Esper

Esper

The Esper component supports the Esper Library[1] for Event Stream Processing. The **camel-esper** library is provided by the Camel Extra[2] project which hosts all *GPL related components for Fuse Mediation Router.

URI format

```
esper:name[?options]
```

When consuming from an Esper endpoint you must specify a **pattern** or **eql** statement to query the event stream.

For example

```
from("esper://cheese?pattern=every event=MyEvent(bar=5)").
 to("activemq:Foo");
```

Options

Name	Default Value	Description
`pattern`		The Esper Pattern expression[3] as a String to filter events
`eql`		The Esper EQL expression[4] as a String to filter events

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

Demo

There is a demo which shows how to work with ActiveMQ, Fuse Mediation Router and Esper[5] in the Camel Extra[6] project

- Esper Fuse Mediation Router Demo[7]

[1] http://esper.codehaus.org
[2] http://code.google.com/p/camel-extra/
[3] http://esper.codehaus.org/esper-1.11.0/doc/reference/en/html/event_patterns.html
[4] http://esper.codehaus.org/esper-1.11.0/doc/reference/en/html/eql_clauses.html
[5] http://code.google.com/p/camel-extra/wiki/EsperDemo
[6] http://code.google.com/p/camel-extra/
[7] http://code.google.com/p/camel-extra/wiki/EsperDemo

Chapter 21. Event

Event Component

The **event:** component provides access to the Spring ApplicationEvent objects. This allows you to publish ApplicationEvent objects to a Spring ApplicationContext or to consume them. You can then use Enterprise Integration Patterns to process them such as Message Filter.

URI format

```
event://default
```

As of FUSE Mediation Router 1.5 the component prefix has been renamed to `spring-event`

```
spring-event://default
```

Chapter 22. EventAdmin

EventAdmin component

Available in Camel 2.6

The `eventadmin` component can be used in an OSGi environment to receive OSGi EventAdmin events and process them.

Dependencies

Maven users need to add the following dependency to their `pom.xml`

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-eventadmin</artifactId>
  <version>${camel-version}</version>
</dependency>
```

where `$\{camel-version\}` must be replaced by the actual version of Camel (2.6.0 or higher).

URI format

```
eventadmin:topic
```

where `topic` is the name of the topic to listen too.

URI options

Name	Default value	Description

Message headers

Name	Type	Message	Description

Message body

The `in` message body will be set to the received Event.

Example usage

```
<route>
    <from uri="eventadmin:*"/>
    <to uri="stream:out"/>
</route>
```

Chapter 23. Exec

Exec component

Available in Fuse Mediation Router 2.3

The `exec` component can be used to execute a system command.

Dependencies

Maven users need to add the following dependency to their `pom.xml`

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-exec</artifactId>
  <version>${camel-version}</version>
</dependency>
```

where `${camel-version}` must be replaced by the actual version of Fuse Mediation Router (2.3.0 or higher).

URI format

```
exec://executable[?options]
```

where `executable` is the name, or file path, of the system command that will be executed. If executable name is used (e.g. `exec:java`), the executable must in the system path.

URI options

Name	Default value	Description
`args`	`null`	The arguments of the executable. The arguments may be one or many whitespace-separated tokens, that can be quoted with `"`, e.g. `args="arg 1" arg2` will use two arguments `arg 1` and `arg2`. To include the quotes use `""`, e.g. `args=""arg 1"" arg2` will use the arguments `"arg 1"` and `arg2`.
`workingDir`	`null`	The directory in which the command should be executed. If `null`, the working directory of the current process will be used.
`timeout`	`Long.MAX_VALUE`	The timeout, in milliseconds, after which the executable should be terminated. If the executable

		has has not finished within the timeout, the component will send a termination request.
`outFile`	`null`	The name of a file, created by the executable, that should be considered as output of the executable. If no `outFile` is set, the standard output (stdout) of the executable will be considered as output.
`binding`	a `DefaultExecBinding` instance	A reference to a `org.apache.commons.exec.ExecBinding` in the Registry[1].
`commandExecutor`	a `DefaultCommandExecutor` instance	A reference to a `org.apache.commons.exec.ExecCommandExecutor` in the Registry[2], that customizes the command execution. The default command executor utilizes the commons-exec library[3]. It adds a shutdown hook for every executed command.
`useStderrOnEmptyStdout`	`false`	A boolean which dictates when `stdin` is empty, it should fallback and use `stderr` in the Message Body. This option is default `false`.

Message headers

The supported headers are defined in `org.apache.camel.component.exec.ExecBinding`.

Name	Type	Message	Description
`ExecBinding.EXEC_COMMAND_EXECUTABLE`	`String`	`in`	The name of the system command that will be executed. Overrides the `executable` in the URI.
`ExecBinding.EXEC_COMMAND_ARGS`	`java.util.List<String>`	`in`	The arguments of the executable. The arguments are

[1] http://camel.apache.org/registry.html
[2] http://camel.apache.org/registry.html
[3] http://commons.apache.org/exec/

			used literally, no quoting is applied. Overrides existing `args` in the URI.
`ExecBinding.EXEC_COMMAND_OUT_FILE`	`String`	`in`	The name of a file, created by the executable, that should be considered as output of the executable. Overrides existing `outFile` in the URI.
`ExecBinding.EXEC_COMMAND_TIMEOUT`	`long`	`in`	The timeout, in milliseconds, after which the executable should be terminated. Overrides existing `timeout` in the URI.
`ExecBinding.EXEC_COMMAND_WORKING_DIR`	`String`	`in`	The directory in which the command should be executed. Overrides existing `workingDir` in the URI.
`ExecBinding.EXEC_EXIT_VALUE`	`int`	`out`	The value of this header is the *exit value* of the executable. Typically not-zero exit

			values indicates abnormal termination. Note that the exit value is OS-dependent.
`ExecBinding.EXEC_STDERR`	`java.io.InputStream`	`out`	The value of this header points to the standard error stream (stderr) of the executable. If no stderr is written, the value is `null`.
`ExecBinding.EXEC_USE_STDERR_ON_EMPTY_STDOUT`	`boolean`	`in`	Indicates when the `stdin` is empty, should we fallback and use `stderr` as the body of the Message. By default this option is `false`.

Message body

If the `in` message body, that that the `Exec` component receives, is convertible to `java.io.InputStream`, it is used to feed input of the executable via its stdin. After the execution, the message body[4] is the result of the execution, that is `org.apache.camel.components.exec.ExecResult` instance containing the stdout, stderr, exit value and out file. The component supports the following `ExecResult` type converters[5] for convenience:

From	To
`ExecResult`	`java.io.InputStream`
`ExecResult`	`String`
`ExecResult`	`byte []`

[4] http://camel.apache.org/exchange.html
[5] http://camel.apache.org/type-converter.html

```
ExecResult org.w3c.dom.Document
```

If out file is used (the endpoint is configured with `outFile`, or there is `ExecBinding.EXEC_COMMAND_OUT_FILE` header) the converters return the content of the out file. If no out file is used, then the converters will use the stdout of the process for conversion to the target type. For example refer to the usage examples.

Executing word count (Linux)

The example below executes `wc` (word count, Linux) to count the words in file `/usr/share/dict/words`. The word count (output) is written in the standart output stream of `wc`.

```
from("direct:exec")
.to("exec:wc?args=--words /usr/share/dict/words")
.process(new Processor() {
     public void process(Exchange exchange) throws Exception {
       // By default, the body is ExecResult instance
       assertIsInstanceOf(ExecResult.class, exchange.getIn().getBody());
       // Use the Camel Exec String type converter to convert the ExecResult to String
       // In this case, the stdout is considered as output
       String wordCountOutput = exchange.getIn().getBody(String.class);
       // do something with the word count
     }
});
```

Executing java

The example below executes `java` with 2 arguments: `-server` and `-version`, provided that `java` is in the system path.

```
from("direct:exec")
.to("exec:java?args=-server -version")
```

The example below executes `java` in `c:/temp` with 3 arguments: `-server`, `-version` and the sytem property `user.name`.

```
from("direct:exec")
.to("exec:c:/program files/jdk/bin/java?args=-server -version -Duser.name=Camel&working
Dir=c:/temp")
```

Executing Ant scripts

The following example executes Apache Ant[6] (Windows only) with the build file `CamelExecBuildFile.xml`, provided that `ant.bat` is in the system path, and that `CamelExecBuildFile.xml` is in the current directory.

```
from("direct:exec")
.to(exec:ant.bat?args=-f CamelExecBuildFile.xml")
```

In the next example, the `ant.bat` command, redirects the ant output to `CamelExecOutFile.txt` with `-l`. The file `CamelExecOutFile.txt` is used as out file with `outFile=CamelExecOutFile.txt`. The example assumes that `ant.bat` is in the system path, and that `CamelExecBuildFile.xml` is in the current directory.

```
from("direct:exec")
.to("exec:ant.bat?args=-f CamelExecBuildFile.xml -l CamelExecOutFile.txt&outFile=CamelEx
ecOutFile.txt")
.process(new Processor() {
     public void process(Exchange exchange) throws Exception {
        InputStream outFile = exchange.getIn().getBody(InputStream.class);
        assertIsInstanceOf(InputStream.class, outFile);
        // do something with the out file here
     }
  });
```

[6] http://ant.apache.org/

Chapter 24. File2

File Component - Fuse Mediation Router 2.0 onwards

The File component provides access to file systems, allowing files to be processed by any other Fuse Mediation Router Components on page 3 or messages from other components to be saved to disk.

URI format

```
file:directoryName[?options]
```

or

```
file://directoryName[?options]
```

Where **directoryName** represents the underlying file directory.

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

Only directories

Fuse Mediation Router 2.0 only support endpoints configured with a starting directory. So the **directoryName** must be a directory. If you want to consume a single file only, you can use the **fileName** option, e.g. by setting `fileName=thefilename`. Also, the starting directory must not contain dynamic expressions with ${ } placeholders. Again use the `fileName` option to specify the dynamic part of the filename.

In Fuse Mediation Router 1.x you could also configure a file and this caused more harm than good as it could lead to confusing situations.

Avoid reading files currently being written by another application

Beware the JDK File IO API is a bit limited in detecting whether another application is currently writing/copying a file. And the implementation can be different depending on OS platform as well. This could lead to that Fuse Mediation Router thinks the file is not locked by another process and start consuming it. Therefore you have to do you own investigation as to what suits your environment. To help with this, Fuse Mediation Router provides different `readLock` options and the `doneFileOption` option that you can use. See also the section "Consuming files from folders where others drop files directly" on page 138.

URI Options

Name	Default Value	Description
`autoCreate`	`true`	Automatically create missing directories in the file's pathname. For the file consumer, that means creating the starting directory. For the file producer, it means the directory to where the files should be written.
`bufferSize`	128kb	Write buffer sized in bytes.
`fileName`	`null`	Use Expression such as File Language to dynamically set the filename. For consumers, it's used as a filename filter. For producers, it's used to evaluate the filename to write. If an expression is set, it take precedence over the `CamelFileName` header. (**Note:** The header itself can also be an Expression). The expression options support both `String` and `Expression` types. If the expression is a `String` type, it is **always** evaluated using the File Language. If the expression is an `Expression` type, the specified `Expression` type is used - this allows you, for instance, to use OGNL expressions. For the consumer, you can use it to filter filenames, so you can for instance consume today's file using the File Language syntax: `mydata-${date:now:yyyyMMdd}.txt`.
`flatten`	`false`	Flatten is used to flatten the file name path to strip any leading paths, so it's just the file name. This allows you to consume recursively into sub-directories, but when you eg write the files to another directory they will be written in a single directory. Setting this to `true` on the producer enforces that any file name recived in `CamelFileName` header will be stripped for any leading paths.
`charset`	`null`	**Camel 2.5:** this option is used to specify the encoding of the file, and camel will set the Exchange property with Exchange.CHARSET_NAME with the value of this option.

Consumer only

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`	Set to `true` to use fixed delay between pools, otherwise fixed rate is used. S ScheduledExecutorService[1] in JDK for details.
`recursive`	`false`	If a directory, will look for files in all the sub-directories as well.

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html

`delete`	`false`	If `true`, the file will be deleted **after** it is processed
`noop`	`false`	If `true`, the file is not moved or deleted in any way. This option is good fo data, or for ETL type requirements. If `noop=true`, Fuse Mediation Router `idempotent=true` as well, to avoid consuming the same files over and o
`preMove`	`null`	Use Expression such as File Language to dynamically set the filename wh it **before** processing. For example to move in-progress files into the `orde` set this value to `order`.
`move`	`.camel`	Use Expression such as File Language to dynamically set the filename wh it **after** processing. To move files into a `.done` subdirectory just enter `.do`
`moveFailed`	`null`	Use Expression such as File Language to dynamically set the filename wh **failed files after** processing. To move files into a `error` subdirectory just er **Note:** When moving the files to another location it can/will **handle** the err you move it to another location so Fuse Mediation Router cannot pick up again.
`include`	`null`	Is used to include files, if filename matches the regex pattern.
`exclude`	`null`	Is used to exclude files, if filename matches the regex pattern.
`idempotent`	`false`	Option to use the Idempotent Consumer EIP pattern to let Fuse Mediatio skip already processed files. Will by default use a memory based LRUCa holds 1000 entries. If `noop=true` then idempotent will be enabled as well consuming the same files over and over again.
`idempotentRepository`	`null`	Pluggable repository as a org.apache.camel.processor.idempotent.MessageIdRepository[2] class. Wil use `MemoryMessageIdRepository` if none is specified and `idempotent` i
`inProgressRepository`	`memory`	Pluggable in-progress repository as a org.apache.camel.processor.idempotent.MessageIdRepository[3] class. The i repository is used to account the current in progress files being consumed. a memory based repository is used.
`filter`	`null`	Pluggable filter as a `org.apache.camel.component.file.GenericFile` class. Will skip files if filter returns `false` in its `accept()` method. Fuse M Router also ships with an **ANT path matcher** filter in the `camel-spring` c More details in section below.
`sorter`	`null`	Pluggable sorter as a java.util.Comparator<org.apache.camel.component.file.GenericFile>[4] cla

[2] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/processor/idempotent/MessageIdRepository.html
[3] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/processor/idempotent/MessageIdRepository.html
[4] http://java.sun.com/j2se/1.5.0/docs/api/java/util/Comparator.html

`sortBy`	`null`	Built-in sort using the File Language. Supports nested sorts, so you can have by file name and as a 2nd group sort by modified date. See sorting section be details.
`readLock`	`markerFile`	Used by consumer, to only poll the files if it has exclusive read-lock on the fil the file is not in-progress or being written). Fuse Mediation Router will wait u file lock is granted. The `readLock` option supports the following built-in strategies: • `markerFile` is the behaviour from Fuse Mediation Router 1.x, where Fuse Me Router will create a marker file and hold a lock on the marker file. This opt **not** available for the FTP on page 161 component. • `changed` uses a length/modification timestamp to detect whether the file is cu being copied or not. Will wait at least 1 second to determine this, so this o cannot consume files as fast as the others, but can be more reliable as the IO API cannot always determine whether a file is currently being used by a process. This option is **not** available for the FTP on page 161 component. • `fileLock` uses `java.nio.channels.FileLock`. This option is **not** availab the FTP on page 161 component. • `rename` attempts to rename the file, in order to test whether we can get an ex read-lock. • `none` is for no read locks at all.
`readLockTimeout`	`0`	Optional timeout in milliseconds for the read-lock, if supported by the read-lo the read-lock could not be granted and the timeout triggered, then Fuse Med Router will skip the file. At next poll Fuse Mediation Router, will try the file aga this time maybe the read-lock could be granted. Currently `fileLock`, `change` `rename` support the timeout.
`readLockCheckInterval`	`1000`	**Camel 2.6:** Interval in millis for the read-lock, if supported by the read lock. T interval is used for sleeping between attempts to acquire the read lock. For ex when using the `changed` read lock, you can set a higher interval period to ca *slow writes*. The default of 1 sec. may be *too fast* if the producer is very slow the file.
`exclusiveReadLockStrategy`	`null`	Pluggable read-lock as a `org.apache.camel.component.file.GenericFileExclusiveReadLockSt` implementation.
`doneFileName`	`null`	**Camel 2.6:** If provided, Camel will only consume files if a *done* file exists. This configures what file name to use. Either you can specify a fixed name. Or yo

		use dynamic placeholders. The *done* file is **always** expected in the same the original file. See *using done file* and *writing done file* sections for exa
`processStrategy`	`null`	A pluggable `org.apache.camel.component.file.GenericFileProcess` allowing you to implement your own `readLock` option or similar. Can also when special conditions must be met before a file can be consumed, such a *ready* file exists. If this option is set then the `readLock` option does not ap
`maxMessagesPerPoll`	`0`	An integer that defines the maximum number of messages to gather per default, no maximum is set. Can be used to set a limit of e.g. 1000 to avo the server read thousands of files as it starts up. Set a value of 0 or nega disabled it.
`startingDirectoryMustExist`	`false`	Whether the starting directory must exist. Mind that the `autoCreate` optio enabled, which means the starting directory is normally auto-created if it do You can disable `autoCreate` and enable this to ensure the starting direct exist. Will throw an exception, if the directory doesn't exist.
`directoryMustExist`	`false`	Similar to `startingDirectoryMustExist` but this applies during polling r sub-directories.

Default behavior for file consumer

- By default the file is locked for the duration of the processing.
- After the route has completed, files are moved into the `.camel` subdirectory, so that they appear to be deleted.
- The File Consumer will always skip any file whose name starts with a dot, such as `.`, `.camel`, `.m2` or `.groovy`.
- Only files (not directories) are matched for valid filename, if options such as: `includeNamePrefix`, `includeNamePostfix`, `excludeNamePrefix`, `excludeNamePostfix`, `regexPattern` are used.

Producer only

Name	Default Value	Description
`fileExist`	`Override`	What to do if a file already exists with the same name. The following values can be specified: **Override**, **Append**, **Fail** and **Ignore**. `Override`, which is the default, replaces the existing file. `Append` adds content to the existing file. `Fail` throws a `GenericFileOperationException`, indicating that there is already an existing file. `Ignore` silently ignores the problem and **does not** override the existing file, but assumes everything is okay.

`tempPrefix`	`null`	This option is used to write the file using a temporary name and then, after the write is complete, rename it to the real name. Can be used to identify files being written and also avoid consumers (not using exclusive read locks) reading in progress files. Is often used by FTP on page 161 when uploading big files.
`tempFileName`	`null`	**Camel 2.1:** The **same** as `tempPrefix` option but offering a more fine grained control on the naming of the temporary filename as it uses the File Language[5].
`keepLastModified`	`false`	**Camel 2.2:** Will keep the last modified timestamp from the source file (if any). Will use the `Exchange.FILE_LAST_MODIFIED` header to located the timestamp. This header can contain either a `java.util.Date` or `long` with the timestamp. If the timestamp exists and the option is enabled it will set this timestamp on the written file. **Note:** This option only applies to the **file** producer. You *cannot* use this option with any of the ftp producers.
`eagerDeleteTargetFile`	`true`	**Camel 2.3:** Whether or not to eagerly delete any existing target file. This option only applies when you use `fileExists=Override` and the `tempFileName` option as well. You can use this to disable (set it to false) deleting the target file before the temp file is written. For example you may write big files and want the target file to exists during the temp file is being written. This ensure the target file is only deleted until the very last moment, just before the temp file is being renamed to the target filename.
`doneFileName`	`null`	**Camel 2.6:** If provided, then Camel will write a 2nd *done* file when the original file has been written. The *done* file will be empty. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders. The *done* file will **always** be written in the same folder as the original file. See *writing done file* section for examples.

Default behavior for file producer

- By default it will override any existing file, if one exist with the same name. In Fuse Mediation Router 1.x the `Append` is the default for the file producer. We have changed this to `Override` in Fuse Mediation Router 2.0 as this is also the default file operation using `java.io.File`. And also the default for the FTP library we use in the camel-ftp on page 161 component.

[5] File Language

Move and Delete operations

Any move or delete operations is executed after (post command) the routing has completed; so during processing of the `Exchange` the file is still located in the inbox folder.

Lets illustrate this with an example:

```
    from("file://inbox?move=.done").to("bean:handleOrder");
```

When a file is dropped in the `inbox` folder, the file consumer notices this and creates a new `FileExchange` that is routed to the `handleOrder` bean. The bean then processes the `File` object. At this point in time the file is still located in the `inbox` folder. After the bean completes, and thus the route is completed, the file consumer will perform the move operation and move the file to the `.done` sub-folder.

The **move** and **preMove** options should be a directory name, which can be either relative or absolute. If relative, the directory is created as a sub-folder from within the folder where the file was consumed.

By default, Fuse Mediation Router will move consumed files to the `.camel` sub-folder relative to the directory where the file was consumed.

If you want to delete the file after processing, the route should be:

```
    from("file://inobox?delete=true").to("bean:handleOrder");
```

We have introduced a **pre** move operation to move files **before** they are processed. This allows you to mark which files have been scanned as they are moved to this sub folder before being processed.

```
    from("file://inbox?preMove=inprogress").to("bean:handleOrder");
```

You can combine the **pre** move and the regular move:

```
    from("file://inbox?preMove=inprogress&move=.done").to("bean:handleOrder");
```

So in this situation, the file is in the `inprogress` folder when being processed and after it's processed, it's moved to the `.done` folder.

Fine grained control over Move and PreMove option

The **move** and **preMove** option is Expression-based, so we have the full power of the File Language[6] to do advanced configuration of the directory and name pattern. Fuse Mediation Router will, in fact, internally convert the directory name you enter into a File Language[7] expression. So when we enter `move=.done` Fuse Mediation Router will convert this into: `${file:parent}/.done/${file:onlyname}`. This is only done if Fuse Mediation

[6] http://camel.apache.org/file-language.html
[7] http://camel.apache.org/file-language.html

Router detects that you have not provided a `${ }` in the option value yourself. So when you enter an expression containing `${ }`, the expression is interpreted as a File Language expression.

So if we want to move the file into a backup folder with today's date as the pattern, we can do:

```
move=backup/${date:now:yyyyMMdd}/${file:name}
```

About moveFailed

The `moveFailed` option allows you to move files that **could not** be processed succesfully to another location such as a error folder of your choice. For example to move the files in an error folder with a timestamp you can use `moveFailed=/error/${file:name.noext}-${date:now:yyyyMMddHHmmssSSS}.${file:name.ext}`.

See more examples at File Language[8].

Message Headers

The following headers are supported by this component:

File producer only

Header	Description
`CamelFileName`	Specifies the name of the file to write (relative to the endpoint directory). The name can be a `String`; a `String` with a File Language or Simple expression; or an Expression object. If it's `null` then Fuse Mediation Router will auto-generate a filename based on the message unique ID.
`CamelFileNameProduced`	The actual absolute filepath (path + name) for the output file that was written. This header is set by Camel and its purpose is providing end-users with the name of the file that was written.

File consumer only

Header	Description
`CamelFileName`	Name of the consumed file as a relative file path with offset from the starting directory configured on the endpoint.
`CamelFileNameOnly`	Only the file name (the name with no leading paths).
`CamelFileAbsolute`	A `boolean` option specifying whether the consumed file denotes an absolute path or not. Should normally be `false` for relative paths. Absolute paths should normally

[8] http://camel.apache.org/file-language.html

	not be used but we added to the move option to allow moving files to absolute paths. But can be used elsewhere as well.
`CamelFileAbsolutePath`	The absolute path to the file. For relative files this path holds the relative path instead.
`CamelFilePath`	The file path. For relative files this is the starting directory + the relative filename. For absolute files this is the absolute path.
`CamelFileRelativePath`	The relative path.
`CamelFileParent`	The parent path.
`CamelFileLength`	A `long` value containing the file size.
`CamelFileLastModified`	A `Date` value containing the last modified timestamp of the file.

Batch Consumer

This component implements the Batch Consumer.

Exchange Properties, file consumer only

As the file consumer is `BatchConsumer` it supports batching the files it polls. By batching it means that Fuse Mediation Router will add some properties to the Exchange so you know the number of files polled the current index in that order.

Property	Description
`CamelBatchSize`	The total number of files that was polled in this batch.
`CamelBatchIndex`	The current index of the batch. Starts from 0.
`CamelBatchComplete`	A `boolean` value indicating the last Exchange in the batch. Is only `true` for the last entry.

This allows you for instance to know how many files exists in this batch and for instance let the Aggregator aggregate this number of files.

Common gotchas with folder and filenames

When Fuse Mediation Router is producing files (writing files) there are a few gotchas affecting how to set a filename of your choice. By default, Fuse Mediation Router will use the message ID as the filename, and since the message ID is normally a unique generated ID, you will end up with filenames such as: `ID-MACHINENAME-2443-1211718892437-1-0`. If such a filename is not desired, then you must provide a filename in the `CamelFileName` message header. The constant, `Exchange.FILE_NAME`, can also be used.

The sample code below produces files using the message ID as the filename:

```
from("direct:report").to("file:target/reports");
```

To use `report.txt` as the filename you have to do:

```
from("direct:report").setHeader(Exchange.FILE_NAME, constant("report.txt")).to( "file:tar
get/reports");
```

Or the same as above, but with `CamelFileName`:

```
from("direct:report").setHeader("CamelFileName", constant("report.txt")).to( "file:target/re
ports");
```

And a syntax where we set the filename on the endpoint with the **fileName** URI option.

```
from("direct:report").to("file:target/reports/?fileName=report.txt");
```

Filename Expression

Filename can be set either using the **expression** option or as a string-based File Language[9] expression in the `CamelFileName` header. See the File Language[10] for syntax and samples.

Consuming files from folders where others drop files directly

Beware if you consume files from a folder where other applications write files directly. Take a look at the different `readLock` options to see what suits your use cases. The best approach is however to write to another folder and after the write move the file in the drop folder. However if you write files directly to the drop folder then the option `changed` could better detect whether a file is currently being written/copied as it uses a file changed algorithm to see whether the file size / modification changes over a period of time. The other read lock options rely on Java File API that sadly is not always very good at detecting this. You may also want to look at the `doneFileName` option, which uses a marker file (done) to signal when a file is done and ready to be consumed.

Using done files

Available as of Camel 2.6

See also section *writing done files* below.

If you want only to consume files when a done file exist, then you can use the `doneFileName` option on the endpoint.

```
    from("file:bar?doneFileName=done");
```

[9] http://camel.apache.org/file-language.html
[10] http://camel.apache.org/file-language.html

```
```

Will only consume files from the bar folder, if a file name done exists in the same directory as the target files. Camel will automatic delete the done file when it's done consuming the files.

However its more common to have one done file per target file. This means there is a 1:1 correlation. To do this you must use dynamic placeholders in the `doneFileName` option. Currently Camel supports the following two dynamic tokens: `file:name` and `file:name.noext` which must be enclosed in ${ }. The consumer only supports the static part of the done file name as either prefix or suffix (not both).

```
    from("file:bar?doneFileName=${file:name}.done");
```

In this example only files will be polled if there exists a done file with the name *file name*.done. For example

- `hello.txt` - is the file to be consumed
- `hello.txt.done` - is the associated done file

You can also use a prefix for the done file, such as:

```
    from("file:bar?doneFileName=ready-${file:name}");
```

- `hello.txt` - is the file to be consumed
- `ready-hello.txt` - is the associated done file

Writing done files

Available as of Camel 2.6

After you have written af file you may want to write an additional *done* file as a kinda of marker, to indicate to others that the file is finished and has been written. To do that you can use the `doneFileName` option on the file producer endpoint.

```
.to("file:bar?doneFileName=done");
```

Will simply create a file named `done` in the same directory as the target file.

However its more common to have one done file per target file. This means there is a 1:1 correlation. To do this you must use dynamic placeholders in the `doneFileName` option. Currently Camel supports the following two dynamic tokens: `file:name` and `file:name.noext` which must be enclosed in `${ }`.

```
.to("file:bar?doneFileName=done-${file:name}");
```

Will for example create a file named `done-foo.txt` if the target file was `foo.txt` in the same directory as the target file.

```
.to("file:bar?doneFileName=${file:name}.done");
```

Will for example create a file named `foo.txt.done` if the target file was `foo.txt` in the same directory as the target file.

```
.to("file:bar?doneFileName=${file:name.noext}.done");
```

Will for example create a file named `foo.done` if the target file was `foo.txt` in the same directory as the target file.

Read from a directory and write to another directory

```
from("file://inputdir/?delete=true").to("file://outputdir")
```

Listen on a directory and create a message for each file dropped there. Copy the contents to the `outputdir` and delete the file in the `inputdir`.

Reading recursive from a directory and write the another

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir")
```

Listen on a directory and create a message for each file dropped there. Copy the contents to the `outputdir` and delete the file in the `inputdir`. Will scan recursively into sub-directories. Will lay out the files in the same directory structure in the `outputdir` as the `inputdir`, including any sub-directories.

```
inputdir/foo.txt
inputdir/sub/bar.txt
```

Will result in the following output layout:

```
outputdir/foo.txt
outputdir/sub/bar.txt
```

Using flatten

If you want to store the files in the outputdir directory in the same directory, disregarding the source directory layout (e.g. to flatten out the path), you just add the `flatten=true` option on the file producer side:

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir?flatten=true")
```

Will result in the following output layout:

```
outputdir/foo.txt
outputdir/bar.txt
```

Reading from a directory and the default move operation

Fuse Mediation Router will by default move any processed file into a `.camel` subdirectory in the directory the file was consumed from.

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir")
```

Affects the layout as follows: **before**

```
inputdir/foo.txt
inputdir/sub/bar.txt
```

after

```
inputdir/.camel/foo.txt
inputdir/sub/.camel/bar.txt
outputdir/foo.txt
outputdir/sub/bar.txt
```

Read from a directory and process the message in java

```
from("file://inputdir/").process(new Processor() {
  public void process(Exchange exchange) throws Exception {
    Object body = exchange.getIn().getBody();
    // do some business logic with the input body
  }
});
```

The body will be a `File` object that points to the file that was just dropped into the `inputdir` directory.

Read files from a directory and send the content to a jms queue

```
from("file://inputdir/").convertBodyTo(String.class).to("jms:test.queue")
```

By default the file endpoint sends a `FileMessage` which contains a `File` object as the body. If you send this directly to the JMS component the JMS message will only contain the `File` object but not the content. By converting the `File` to a `String`, the message will contain the file contents what is probably what you want.

The route above using Spring DSL:

```
   <route>
      <from uri="file://inputdir/"/>
      <convertBodyTo type="java.lang.String"/>
      <to uri="jms:test.queue"/>
   </route>
```

Writing to files

Fuse Mediation Router is of course also able to write files, i.e. produce files. In the sample below we receive some reports on the SEDA queue that we processes before they are written to a directory.

```
public void testToFile() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedFileExists("target/test-reports/report.txt");

    template.sendBody("direct:reports", "This is a great report");

    assertMockEndpointsSatisfied();
}

protected JndiRegistry createRegistry() throws Exception {
    // bind our processor in the registry with the given id
    JndiRegistry reg = super.createRegistry();
    reg.bind("processReport", new ProcessReport());
    return reg;
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // the reports from the seda queue is processed by our processor
            // before they are written to files in the target/reports directory
            from("direct:reports").processRef("processReport").to("file://target/test-re
ports", "mock:result");
        }
    };
}

private class ProcessReport implements Processor {

    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        // do some business logic here

        // set the output to the file
        exchange.getOut().setBody(body);
```

```
       // set the output filename using java code logic, notice that this is done by setting

        // a special header property of the out exchange
        exchange.getOut().setHeader(Exchange.FILE_NAME, "report.txt");
    }

}
```

Write to subdirectory using Exchange.FILE_NAME

Using a single route, it is possible to write a file to any number of subdirectories. If you have a route setup as such:

```
  <route>
    <from uri="bean:myBean"/>
    <to uri="file:/rootDirectory"/>
  </route>
```

You can have `myBean` set the header `Exchange.FILE_NAME` to values such as:

```
Exchange.FILE_NAME = hello.txt => /rootDirectory/hello.txt
Exchange.FILE_NAME = foo/bye.txt => /rootDirectory/foo/bye.txt
```

This allows you to have a single route to write files to multiple destinations.

Using expression for filenames

In this sample we want to move consumed files to a backup folder using today's date as a sub-folder name:

```
from("file://inbox?move=backup/${date:now:yyyyMMdd}/${file:name}").to("...");
```

See File Language for more samples.

Avoiding reading the same file more than once (idempotent consumer)

Fuse Mediation Router supports Idempotent Consumer directly within the component so it will skip already processed files. This feature can be enabled by setting the `idempotent=true` option.

```
from("file://inbox?idempotent=true").to("...");
```

By default Fuse Mediation Router uses a in memory based store for keeping track of consumed files, it uses a least recently used cache storing holding up to 1000 entries. You can plugin your own implementation of this store by using the `idempotentRepository` option using the `#` sign in the value to indicate it's a referring to a bean in the Registry with the specified `id`.

```
   <!-- define our store as a plain spring bean -->
   <bean id="myStore" class="com.mycompany.MyIdempotentStore"/>
```

```
  <route>
    <from uri="file://inbox?idempotent=true&dempotentRepository=#myStore"/>
    <to uri="bean:processInbox"/>
  </route>
```

Fuse Mediation Router will log at `DEBUG` level if it skips a file because it has been consumed before:

```
DEBUG FileConsumer is idempotent and the file has been consumed before. Will skip this file:
 target\idempotent\report.txt
```

Using a file based idempotent repository

In this section we will use the file based idempotent repository `org.apache.camel.processor.idempotent.FileIdempotentRepository` instead of the in-memory based that is used as default. This repository uses a 1st level cache to avoid reading the file repository. It will only use the file repository to store the content of the 1st level cache. Thereby the repository can survive server restarts. It will load the content of the file into the 1st level cache upon startup. The file structure is very simple as it store the key in separate lines in the file. By default, the file store has a size limit of 1mb when the file grew larger Fuse Mediation Router will truncate the file store be rebuilding the content by flushing the 1st level cache in a fresh empty file.

We configure our repository using Spring XML creating our file idempotent repository and define our file consumer to use our repository with the `idempotentRepository` using `\#` sign to indicate Registry lookup:

```
<!-- this is our file based idempotent store configured to use the .filestore.dat as file
-->
<bean id="fileStore" class="org.apache.camel.processor.idempotent.FileIdempotentRepository">

    <!-- the filename for the store -->
    <property name="fileStore" value="target/fileidempotent/.filestore.dat"/>
    <!-- the max filesize in bytes for the file. Fuse Mediation Router will trunk and flush
 the cache
         if the file gets bigger -->
    <property name="maxFileStoreSize" value="512000"/>
    <!-- the number of elements in our store -->
    <property name="cacheSize" value="250"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file://target/fileidempotent/?idempotent=true&dempotentReposit
ory=#fileStore&ove=done/${file:name}"/>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

Using a JPA based idempotent repository

In this section we will use the JPA based idempotent repository instead of the in-memory based that is used as default.

First we need a persistence-unit in `META-INF/persistence.xml` where we need to use the class `org.apache.camel.processor.idempotent.jpa.MessageProcessed` as model.

```
<persistence-unit name="idempotentDb" transaction-type="RESOURCE_LOCAL">
  <class>org.apache.camel.processor.idempotent.jpa.MessageProcessed</class>

  <properties>
    <property name="openjpa.ConnectionURL" value="jdbc:derby:target/idempotentTest;cre
ate=true"/>
    <property name="openjpa.ConnectionDriverName" value="org.apache.derby.jdbc.Embedded
Driver"/>
    <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema"/>
    <property name="openjpa.Log" value="DefaultLevel=WARN, Tool=INFO"/>
  </properties>
</persistence-unit>
```

Then we need to setup a Spring `jpaTemplate` in the spring XML file:

```
<!-- this is standard spring JPA configuration -->
<bean id="jpaTemplate" class="org.springframework.orm.jpa.JpaTemplate">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalEntityManagerFact
oryBean">
    <!-- we use idempotentDB as the persitence unit name defined in the persistence.xml
file -->
    <property name="persistenceUnitName" value="idempotentDb"/>
</bean>
```

And finally we can create our JPA idempotent repository in the spring XML file as well:

```
<!-- we define our jpa based idempotent repository we want to use in the file consumer -->
<bean id="jpaStore" class="org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository">

    <!-- Here we refer to the spring jpaTemplate -->
    <constructor-arg index="0" ref="jpaTemplate"/>
    <!-- This 2nd parameter is the name  (= a cateogry name).
         You can have different repositories with different names -->
    <constructor-arg index="1" value="FileConsumer"/>
</bean>
```

And then we just need to reference the **jpaStore** bean in the file consumer endpoint, using the `idempotentRepository` option and the `#` syntax:

```
  <route>
    <from uri="file://inbox?idempotent=true&dempotentRepository=#jpaStore"/>
    <to uri="bean:processInbox"/>
  </route>
```

Filter using org.apache.camel.component.file.GenericFileFilter

Fuse Mediation Router supports pluggable filtering strategies. You can then configure the endpoint with such a filter to skip certain files being processed.

In the sample we have build our own filter that skips files starting with `skip` in the filename:

```
public class MyFileFilter implements GenericFileFilter {
    public boolean accept(GenericFile pathname) {
        // we dont accept any files starting with skip in the name
        return !pathname.getFileName().startsWith("skip");
    }
}
```

And then we can configure our route using the **filter** attribute to reference our filter (using `#` notation) that we have defined in the spring XML file:

```
   <!-- define our sorter as a plain spring bean -->
   <bean id="myFilter" class="com.mycompany.MyFileSorter"/>

  <route>
    <from uri="file://inbox?filter=#myFilter"/>
    <to uri="bean:processInbox"/>
  </route>
```

Filtering using ANT path matcher

The ANT path matcher is shipped out-of-the-box in the **camel-spring** jar. So you need to depend on **camel-spring** if you are using Maven. The reasons is that we leverage Spring's AntPathMatcher[11] to do the actual matching.

The file paths is matched with the following rules:

- `?` matches one character
- `*` matches zero or more characters

[11] http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/util/AntPathMatcher.html

- ** matches zero or more directories in a path

The sample below demonstrates how to use it:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <template id="camelTemplate"/>

    <!-- use myFilter as filter to allow setting ANT paths for which files to scan for -->

   <endpoint id="myFileEndpoint" uri="file://target/antpathmatcher?recursive=true&ilter=#my
AntFilter"/>

    <route>
        <from ref="myFileEndpoint"/>
        <to uri="mock:result"/>
    </route>
</camelContext>

<!-- we use the antpath file filter to use ant paths for includes and exlucde -->
<bean id="myAntFilter" class="org.apache.camel.component.file.AntPathMatcherGenericFileFil
ter">
    <!-- include and file in the subfolder that has day in the name -->
    <property name="includes" value="**/subfolder/**/*day*"/>
    <!-- exclude all files with bad in name or .xml files. Use comma to seperate multiple
excludes -->
    <property name="excludes" value="**/*bad*,**/*.xml"/>
</bean>
```

Sorting using Comparator

Fuse Mediation Router supports pluggable sorting strategies. This strategy it to use the build in `java.util.Comparator` in Java. You can then configure the endpoint with such a comparator and have Fuse Mediation Router sort the files before being processed.

In the sample we have built our own comparator that just sorts by file name:

```
public class MyFileSorter implements Comparator<GenericFile> {
    public int compare(GenericFile o1, GenericFile o2) {
        return o1.getFileName().compareToIgnoreCase(o2.getFileName());
    }
}
```

And then we can configure our route using the **sorter** option to reference to our sorter (`mySorter`) we have defined in the spring XML file:

```
   <!-- define our sorter as a plain spring bean -->
   <bean id="mySorter" class="com.mycompany.MyFileSorter"/>
```

```
<route>
  <from uri="file://inbox?sorter=#mySorter"/>
  <to uri="bean:processInbox"/>
</route>
```

URI options can reference beans using the # syntax

In the Spring DSL route about notice that we can reference beans in the Registry by prefixing the id with `#`. So writing `sorter=#mySorter`, will instruct Fuse Mediation Router to go look in the Registry for a bean with the ID, `mySorter`.

Sorting using sortBy

Fuse Mediation Router supports pluggable sorting strategies. This strategy it to use the File Language to configure the sorting. The `sortBy` option is configured as follows:

```
sortBy=group 1;group 2;group 3;...
```

Where each group is separated with semi colon. In the simple situations you just use one group, so a simple example could be:

```
sortBy=file:name
```

This will sort by file name, you can reverse the order by prefixing `reverse:` to the group, so the sorting is now Z..A:

```
sortBy=reverse:file:name
```

As we have the full power of File Language we can use some of the other parameters, so if we want to sort by file size we do:

```
sortBy=file:length
```

You can configure to ignore the case, using `ignoreCase:` for string comparison, so if you want to use file name sorting but to ignore the case then we do:

```
sortBy=ignoreCase:file:name
```

You can combine ignore case and reverse, however reverse must be specified first:

```
sortBy=reverse:ignoreCase:file:name
```

In the sample below we want to sort by last modified file, so we do:

```
sortBy=file:modifed
```

And then we want to group by name as a 2nd option so files with same modifcation is sorted by name:

```
sortBy=file:modifed;file:name
```

Now there is an issue here, can you spot it? Well the modified timestamp of the file is too fine as it will be in milliseconds, but what if we want to sort by date only and then subgroup by name? Well as we have the true power of File Language[12] we can use the its date command that supports patterns. So this can be solved as:

```
sortBy=date:file:yyyyMMdd;file:name
```

Yeah, that is pretty powerful, oh by the way you can also use reverse per group, so we could reverse the file names:

```
sortBy=date:file:yyyyMMdd;reverse:file:name
```

Using GenericFileProcessStrategy

The option `processStrategy` can be used to use a custom `GenericFileProcessStrategy` that allows you to implement your own *begin*, *commit* and *rollback* logic. For instance lets assume a system writes a file in a folder you should consume. But you should not start consuming the file before another *ready* file have been written as well.

So by implementing our own `GenericFileProcessStrategy` we can implement this as:

- In the `begin()` method we can test whether the special *ready* file exists. The begin method returns a `boolean` to indicate if we can consume the file or not.
- in the `commit()` method we can move the actual file and also delete the *ready* file.

Debug logging

This component has log level **TRACE** that can be helpful if you have problems.

See also:

- File Language[13]
- FTP2 on page 161

[12] http://camel.apache.org/file-language.html
[13] http://camel.apache.org/file-language.html

Chapter 25. FIX

FIX

The FIX component supports the FIX protocol[1] by using the QuickFix/J library[2].

URI format

```
fix://configurationResource
```

Where **configurationResource** points to the QuickFix/J configuration file to define how to connect to FIX. This could be a resource on the classpath or a reference to a full URL using the `http:` or `file:` schemes.

Message Formats

By default this component will attempt to use the Type Converter[3] to turn the inbound message body into a QuickFix Message class[4] and all outputs from FIX will be in the same format.

[1] http://en.wikipedia.org/wiki/FIX_protocol
[2] http://www.quickfixj.org/
[3] Type Converter
[4] http://www.quickfixj.org/quickfixj/javadoc/quickfix/Message.html

Chapter 26. Flatpack

Flatpack Component

The Flatpack component supports fixed width and delimited file parsing using the FlatPack library[1]. **Notice:** This component only supports consuming from flatpack files to Object model. You can not (yet) write from Object model to flatpack format.

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

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

URI format

```
flatpack:[delim|fixed]:flatPackConfig.pzmap.xml[?options]
```

Or for a delimited file handler with no configuration file just use:

```
flatpack:someName[?options]
```

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

URI Options

Name	Default Value	Description
`delimiter`	`,`	The default character delimiter for delimited files.
`textQualifier`	`"`	The text qualifier for delimited files.
`ignoreFirstRecord`	`true`	Whether the first line is ignored for delimited files (for the column headers).
`splitRows`	`true`	As of Fuse Mediation Router 1.5, the component can either process each row one by one or the entire content at once.

[1] http://flatpack.sourceforge.net

Examples

- `flatpack:fixed:foo.pzmap.xml` creates a fixed-width endpoint using the `foo.pzmap.xml` file configuration.
- `flatpack:delim:bar.pzmap.xml` creates a delimited endpoint using the `bar.pzmap.xml` file configuration.
- `flatpack:foo` creates a delimited endpoint called `foo` with no file configuration.

Message Headers

Fuse Mediation Router will store the following headers on the IN message:

Header	Description
`camelFlatpackCounter`	The current row index. For `splitRows=false` the counter is the total number of rows.

Message Body

The component delivers the data in the IN message as a `org.apache.camel.component.flatpack.DataSetList` object that has converters for `java.util.Map` or `java.util.List`. Usually you want the `Map` if you process one row at a time (`splitRows=true`). Use `List` for the entire content (`splitRows=false`), where each element in the list is a `Map`. Each `Map` contains the key for the column name and its corresponding value.

For example to get the firstname from the sample below:

```
Map row = exchange.getIn().getBody(Map.class);
String firstName = row.get("FIRSTNAME");
```

However, you can also always get it as a `List` (even for `splitRows=true`). The same example:

```
List data = exchange.getIn().getBody(List.class);
Map row = (Map)data.get(0);
String firstName = row.get("FIRSTNAME");
```

Header and Trailer records

In Fuse Mediation Router 1.5 onwards the header and trailer notions in Flatpack are supported. However, you **must** use fixed record IDs:

- `header` for the header record (must be lowercase)

- `trailer` for the trailer record (must be lowercase)

The example below illustrates this fact that we have a header and a trailer. You can omit one or both of them if not needed.

```
    <RECORD id="header" startPosition="1" endPosition="3" indicator="HBT">
        <COLUMN name="INDICATOR" length="3"/>
        <COLUMN name="DATE" length="8"/>
    </RECORD>

    <COLUMN name="FIRSTNAME" length="35" />
    <COLUMN name="LASTNAME" length="35" />
    <COLUMN name="ADDRESS" length="100" />
    <COLUMN name="CITY" length="100" />
    <COLUMN name="STATE" length="2" />
    <COLUMN name="ZIP" length="5" />

    <RECORD id="trailer" startPosition="1" endPosition="3" indicator="FBT">
        <COLUMN name="INDICATOR" length="3"/>
        <COLUMN name="STATUS" length="7"/>
    </RECORD>
```

Using the endpoint

A common use case is sending a file to this endpoint for further processing in a separate route. For example:

```
  <camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
    <route>
      <from uri="file://someDirectory"/>
      <to uri="flatpack:foo"/>
    </route>

    <route>
      <from uri="flatpack:foo"/>
      ...
    </route>
  </camelContext>
```

You can also convert the payload of each message created to a `Map` for easy Bean Integration

Chapter 27. Freemarker

Freemarker

Available as of Fuse Mediation Router 1.6

The **freemarker:** component allows you to process a message using a Freemarker[1] template. This can be ideal when using Templating to generate responses for requests.

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

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

URI format

```
freemarker:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (for example, `file://folder/myfile.ftl`).

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

Options

Option	Default	Description
`contentCache`	`true`	Cache for the resource content when its loaded.
`encoding`	`null`	Character encoding of the resource content.

Freemarker Context

Fuse Mediation Router will provide exchange information in the Freemarker context (just a `Map`). The `Exchange` is transfered as:

[1] http://freemarker.org/

Key	Value
`exchange`	The Exchange itself.
`headers`	The headers of the In message.
`camelContext`	The Camel Context.
`request`	The In message.
`body`	The In message body.
`response`	The Out message (only for InOut message exchange pattern).

Hot reloading

The Freemarker template resource is by default **not** hot reloadable for both file and classpath resources (expanded jar). If you set `contentCache=false`, then Fuse Mediation Router will not cache the resource and hot reloading is thus enabled. This scenario can be used in development.

Dynamic templates

Available as of Camel 2.1 Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

Header	Type	Description
CamelFreemarkerResourceUri	String	**Camel 2.1:** A URI for the template resource to use instead of the endpoint configured.
CamelFreemarkerTemplate	String	**Camel 2.1:** The template to use instead of the endpoint configured.

Samples

For example, you can define a route like the following:

```
from("activemq:My.Queue").
  to("freemarker:com/acme/MyResponse.ftl");
```

To use a Freemarker template to formulate a response to *InOut* message exchanges (where there is a `JMSReplyTo` header).

If you want to process *InOnly* exchanges, you could use a Freemarker template to transform the message before sending it on to another endpoint:

```
from("activemq:My.Queue").
  to(ExchangePattern.InOut,"freemarker:com/acme/MyResponse.ftl").
  to("activemq:Another.Queue");
```

And to disable the content cache (for example, for development usage where the `.ftl` template should be hot reloaded):

```
from("activemq:My.Queue").
  to(ExchangePattern.InOut,"freemarker:com/acme/MyResponse.ftl?contentCache=false").
  to("activemq:Another.Queue");
```

And for a file-based resource:

```
from("activemq:My.Queue").
  to(ExchangePattern.InOut,"freemarker:file://myfolder/MyResponse.ftl?contentCache=false").

  to("activemq:Another.Queue");
```

In **Camel 2.1** it's possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").
  setHeader("CamelFreemarkerResourceUri").constant("path/to/my/template.ftl").
  to("freemarker:dummy");
```

The Email Sample

In this sample we want to use Freemarker templating for an order confirmation email. The email template is laid out in Freemarker as:

```
Dear ${headers.lastName}, ${headers.firstName}

Thanks for the order of ${headers.item}.

Regards Camel Riders Bookstore
${body}
```

And the java code:

```
private Exchange createLetter() {
    Exchange exchange = context.getEndpoint("direct:a").createExchange();
    Message msg = exchange.getIn();
    msg.setHeader("firstName", "Claus");
    msg.setHeader("lastName", "Ibsen");
    msg.setHeader("item", "Camel in Action");
    msg.setBody("PS: Next beer is on me, James");
```

```
    return exchange;
}

@Test
public void testFreemarkerLetter() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedBodiesReceived("Dear Ibsen, Claus\n\nThanks for the order of Camel in Ac
tion.\n\nRegards Camel Riders Bookstore\nPS: Next beer is on me, James");

    template.send("direct:a", createLetter());

    mock.assertIsSatisfied();
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:a").to("freemarker:org/apache/camel/component/freemarker/let
ter.ftl").to("mock:result");
        }
    };
}
```

Chapter 28. FTP2

FTP/SFTP Component - Fuse Mediation Router 2.0 onwards

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

Using Fuse Mediation Router 1.x

If you are using Fuse Mediation Router 1.x then see this link for documentation. This page is only for Fuse Mediation Router 2.0 or newer.

Using FTPS

The FTPS component is only available in Camel 2.2 or newer. FTPS (also known as FTP Secure) is an extension to FTP that adds support for the Transport Layer Security (TLS) and the Secure Sockets Layer (SSL) cryptographic protocols.

Libraries used

This component uses two different libraries for the actual FTP work. FTP and FTPS uses Apache Commons Net[1] while SFTP uses JCraft JSCH[2].

URI format

```
ftp://[username@]hostname[:port]/directoryname[?options]
sftp://[username@]hostname[:port]/directoryname[?options]
ftps://[username@]hostname[:port]/directoryname[?options]
```

Where **directoryname** represents the underlying directory. Can contain nested folders.

If no **username** is provided, then `anonymous` login is attempted using no password. If no **port** number is provided, Fuse Mediation Router will provide default values according to the protocol (ftp = 21, sftp = 22, ftps = 21).

[1] http://commons.apache.org/net/
[2] http://www.jcraft.com/jsch/

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

URI Options

The options below are exclusive for the FTP2 on page 161 component.

Name	Default Value	Description
`username`	`null`	Specifies the username to use to log in to the remote file systen.
`password`	`null`	Specifies the password to use to log in to the remote file system.
`binary`	`false`	Specifies the file transfer mode, BINARY or ASCII. Default is ASCII (`false`).
`disconnect`	`false`	**Camel 2.2:** Whether or not to disconnect from remote FTP server right after use. Can be used for both consumer and producer. Disconnect will only disconnect the current connection to the FTP server. If you have a consumer which you want to stop, then you need to stop the consumer/route instead.
`localWorkDirectory`	`null`	When consuming, a local work directory can be used to store the remote file content directly in local files, to avoid loading the content into memory. This is beneficial, if you consume a very big remote file and thus can conserve memory. See below for more details.
`passiveMode`	`false`	**FTP only**: Specifies whether to use passive mode connections. Default is active mode {`false`).
`securityProtocol`	`TLS`	**FTPS only:** Sets the underlying security protocol. The following values are defined: `TLS`: Transport Layer Security `SSL`: Secure Sockets Layer
`disableSecureDataChannelDefaults`	`false`	**Camel 2.4: FTPS only**: Whether or not to disable using default values for `execPbsz` and `execProt` when using secure data transfer. You can set this option to `true` if you want to be in absolute full control what the options `execPbsz` and `execProt` should be used.
`execProt`	`null`	**Camel 2.4: FTPS only**: Will by default use option `P` if secure data channel defaults hasn't been disabled. Possible values are: `C`: Clear `S`: Safe (SSL protocol only) `E`: Confidential (SSL protocol only) `P`: Private

execPbsz	null	**Camel 2.4: FTPS only**: This option specifies the buffer size of the secure data channel. If option useSecureDataChannel has been enabled and this option has not been explicit set, then value 0 is used.
isImplicit	false	**FTPS only:** Sets the security mode(implicit/explicit). Default is explicit (false).
knownHostsFile	null	**SFTP only:** Sets the known_hosts file, so that the SFTP endpoint can do host key verification.
privateKeyFilePassphrase	null	**SFTP only:** Set the private key file passphrase to that the SFTP endpoint can do private key verification.
privateKeyFilePassphrase	null	**SFTP only:** Set the private key file passphrase to that the SFTP endpoint can do private key verification.
strictHostKeyChecking	no	**SFTP only:Camel 2.2:** Sets whether to use strict host key checking. Possible values are: no, yes and ask. ask does not make sense to use as Camel cannot answer the question for you as its meant for human intervention. **Note:** The default in Camel 2.1 and below was ask.
maximumReconnectAttempts	3	Specifies the maximum reconnect attempts Fuse Mediation Router performs when it tries to connect to the remote FTP server. Use 0 to disable this behavior.
reconnectDelay	1000	Delay in millis Fuse Mediation Router will wait before performing a reconnect attempt.
connectTimeout	10000	**Camel 2.4:** Is the connect timeout in millis. This corresponds to using ftpClient.connectTimeout for the FTP/FTPS. For SFTP this option is also used when attempting to connect.
soTimeout	null	**FTP and FTPS Only:Camel 2.4:** Is the SocketOptions.SO_TIMEOUT value in millis. Note SFTP will automatic use the connectTimeout as the soTimeout.
timeout	30000	**FTP and FTPS Only:Camel 2.4:** Is the data timeout in millis. This corresponds to using ftpClient.dataTimeout for the FTP/FTPS. For SFTP there is no data timeout.
throwExceptionOnConnectFailed	false	**Camel 2.5:** Whether or not to thrown an exception if a successful connection and login could not be establish. This allows a custom pollStrategy to deal with the exception, for example to stop the consumer or the likes.
siteCommand	null	**FTP and FTPS Only:Camel 2.5:** To execute site commands after successful login. Multiple site commands can be separated using a new line character (\n). Use

		`help site` to see which site commands your FTP server supports.
`stepwise`	`true`	When consuming directories, specifies whether or not to use stepwise mode for traversing the directory tree. Stepwise means that it will CD one directory at a time. For more details, see "Stepwise changing directories" on page 166.
`separator`	`Auto`	**Camel 2.6:** Dictates what path separator `char` to use when uploading files. `Auto` means use the path provided without altering it. `UNIX` means use UNIX style path separators. `Windows` means use Windows style path separators.
`ftpClient`	`null`	**FTP and FTPS Only:Camel 2.1:** Allows you to use a custom `org.apache.commons.net.ftp.FTPClient` instance.
`ftpClientConfig`	`null`	**FTP and FTPS Only:Camel 2.1:** Allows you to use a custom `org.apache.commons.net.ftp.FTPClientConfig` instance.
`ftpClient.trustStore.file`	`null`	**FTPS Only:** Sets the trust store file, so that the FTPS client can look up for trusted certificates.
`ftpClient.trustStore.type`	`JKS`	**FTPS Only:** Sets the trust store type.
`ftpClient.trustStore.algorithm`	`SunX509`	**FTPS Only:** Sets the trust store algorithm.
`ftpClient.trustStore.password`	`null`	**FTPS Only:** Sets the trust store password.
`ftpClient.keyStore.file`	`null`	**FTPS Only:** Sets the key store file, so that the FTPS client can look up for the private certificate.
`ftpClient.keyStore.type`	`JKS`	**FTPS Only:** Sets the key store type.
`ftpClient.keyStore.algorithm`	`SunX509`	**FTPS Only:** Sets the key store algorithm.
`ftpClient.keyStore.password`	`null`	**FTPS Only:** Sets the key store password.
`ftpClient.keyStore.keyPassword`	`null`	**FTPS Only:** Sets the private key password.

FTPS component default trust store

By default, the FTPS component trust store accept all certificates. If you only want trust selective certificates, you have to configure the trust store with the `ftpClient.trustStore.xxx` options or by configuring a custom `ftpClient`.

You can configure additional options on the `ftpClient` and `ftpClientConfig` from the URI directly by using the `ftpClient.` or `ftpClientConfig.` prefix.

For example to set the `setDataTimeout` on the `FTPClient` to 30 seconds you can do:

```
from("ftp://foo@myserver?password=secret&ftpClient.dataTimeout=30000").to("bean:foo");
```

You can mix and match and have use both prefixes, for example to configure date format or timezones.

```
from("ftp://foo@myserver?password=secret&ftpClient.dataTimeout=30000&ftpClientConfig.server
LanguageCode=fr").to("bean:foo");
```

You can have as many of these options as you like.

See the documentation of the Apache Commons FTP FTPClientConfig[3] for possible options and more details. And as well for Apache Commons FTP FTPClient[4].

If you do not like having many and long configuration in the url you can refer to the `ftpClient` or `ftpClientConfig` to use by letting Camel lookup in the Registry for it.

For example:

```
  <bean id="myConfig" class="org.apache.commons.net.ftp.FTPClientConfig">
      <property name="lenientFutureDates" value="true"/>
      <property name="serverLanguageCode" value="fr"/>
  </bean>
```

And then let Camel lookup this bean when you use the # notation in the url.

```
from("ftp://foo@myserver?password=secret&ftpClientConfig=#myConfig").to("bean:foo");
```

[3] http://commons.apache.org/net/api/org/apache/commons/net/ftp/FTPClientConfig.html
[4] http://commons.apache.org/net/api/org/apache/commons/net/ftp/FTPClient.html

More URI options

Important

See File2 on page 129 as all the options there also applies for this component.

Stepwise changing directories

Camel FTP on page 161 can operate in two modes in terms of traversing directories when consuming files (for example, downloading) or producing files (for example, uploading):

- stepwise
- not stepwise

You may want to pick either one depending on your situation and security issues. Some Camel end users can only download files if they use stepwise, while others can only download if they do not. At least you have the choice to pick.

Note that stepwise changing of directory will in most cases only work when the user is confined to it's home directory and when the home directory is reported as `/`.

The difference between the two of them is best illustrated with an example. Suppose we have the following directory structure on the remote FTP server we need to traverse and download files:

```
/
/one
/one/two
/one/two/sub-a
/one/two/sub-b
```

And that we have a file in each of `sub-a` (`a.txt`) and `sub-b` (`b.txt`) folder.

Using stepwise=true (default mode)

The following log shows the conversation between the FTP endpoint and the remote FTP server when the FTP endpoint is operating in *stepwise* mode:

```
TYPE A
200 Type set to A
PWD
```

```
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
SYST
215 UNIX emulated by FileZilla
PORT 127,0,0,1,17,94
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CWD sub-a
250 CWD successful. "/one/two/sub-a" is current directory.
PORT 127,0,0,1,17,95
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CDUP
200 CDUP successful. "/one/two" is current directory.
CWD sub-b
250 CWD successful. "/one/two/sub-b" is current directory.
PORT 127,0,0,1,17,96
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CDUP
200 CDUP successful. "/one/two" is current directory.
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
PORT 127,0,0,1,17,97
200 Port command successful
RETR foo.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
```

```
CWD two
250 CWD successful. "/one/two" is current directory.
CWD sub-a
250 CWD successful. "/one/two/sub-a" is current directory.
PORT 127,0,0,1,17,98
200 Port command successful
RETR a.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
CWD sub-b
250 CWD successful. "/one/two/sub-b" is current directory.
PORT 127,0,0,1,17,99
200 Port command successful
RETR b.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
QUIT
221 Goodbye
disconnected.
```

As you can see when stepwise is enabled, it will traverse the directory structure using CD xxx.

Using stepwise=false

The following log shows the conversation between the FTP endpoint and the remote FTP server when the FTP endpoint is operating in *non-stepwise* mode:

```
230 Logged on
TYPE A
200 Type set to A
SYST
215 UNIX emulated by FileZilla
PORT 127,0,0,1,4,122
200 Port command successful
LIST one/two
150 Opening data channel for directory list
```

```
226 Transfer OK
PORT 127,0,0,1,4,123
200 Port command successful
LIST one/two/sub-a
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,124
200 Port command successful
LIST one/two/sub-b
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,125
200 Port command successful
RETR one/two/foo.txt
150 Opening data channel for file transfer.
226 Transfer OK
PORT 127,0,0,1,4,126
200 Port command successful
RETR one/two/sub-a/a.txt
150 Opening data channel for file transfer.
226 Transfer OK
PORT 127,0,0,1,4,127
200 Port command successful
RETR one/two/sub-b/b.txt
150 Opening data channel for file transfer.
226 Transfer OK
QUIT
221 Goodbye
disconnected.
```

As you can see when not using stepwise, there are no CD operation invoked at all.

Examples

FTP Consumer does not support concurrency

The FTP consumer (with the same endpoint) does not support concurrency (the backing FTP client is not thread safe). You can use multiple FTP consumers to poll from different endpoints. It is only a single endpoint that does not support concurrent consumers.

The FTP producer does **not** have this issue, it supports concurrency.

In the future we will add consumer pooling to Fuse Mediation Router[5] to allow this consumer to support concurrency as well.

More information

This component is an extension of the File2 on page 129 component. So there are more samples and details on the File2 on page 129 component page.

Default when consuming files

The FTP on page 161 consumer will by default leave the consumed files untouched on the remote FTP server. You have to configure it explicit if you want it to delete the files or move them to another location. For example you can use `delete=true` to delete the files, or use `move=.done` to move the files into a hidden done sub directory.

The regular File on page 129 consumer is different as it will by default move files to a `.camel` sub directory. The reason Camel does **not** do this by default for the FTP consumer is that it may lack permissions by default to be able to move or delete files.

limitations

The option **readLock** can be used to force Fuse Mediation Router **not** to consume files that are currently in the process of being written. However, this option is turned off by default, as it requires that the user has write access. There are other solutions to avoid consuming files that are currently being written over FTP; for instance, you can write to a temporary destination and move the file after it has been written.

The ftp producer does **not** support appending to existing files. Any existing files on the remote server will be deleted before the file is written.

[5] https://issues.apache.org/activemq/browse/CAMEL-1682

Message Headers

The following message headers can be used to affect the behavior of the component

Header	Description
`CamelFileName`	Specifies the output file name (relative to the endpoint directory) to be used for the output message when sending to the endpoint. If this is not present and no expression either, then a generated message ID is used as the filename instead.
`CamelFileNameProduced`	The actual absolute filepath (path + name) for the output file that was written. This header is set by Fuse Mediation Router and its purpose is providing end-users the name of the file that was written.
`CamelFileBatchIndex`	Current index out of total number of files being consumed in this batch.
`CamelFileBatchSize`	Total number of files being consumed in this batch.
`CamelFileHost`	The remote hostname.
`CamelFileLocalWorkPath`	Path to the local work file, if local work directory is used.

About timeouts

The two sets of libraries (see above) have different APIs for setting the timeout. You can use the `connectTimeout` option for both of them to set a timeout in milliseconds to establish a network connection. An individual `soTimeout` can also be set on the FTP/FTPS, which corresponds to using `ftpClient.soTimeout`. Notice SFTP will automatically use `connectTimeout` as its `soTimeout`. The `timeout` option only applies for FTP/FTSP as the data timeout, which corresponds to the `ftpClient.dataTimeout` value. All timeout values are in milliseconds.

Using Local Work Directory

Fuse Mediation Router supports consuming from remote FTP servers and downloading the files directly into a local work directory. This avoids reading the entire remote file content into memory as it is streamed directly into the local file using `FileOutputStream`.

Fuse Mediation Router will store to a local file with the same name as the remote file, though with `.inprogress` as extension while the file is being downloaded. Afterwards, the file is renamed to remove the `.inprogress` suffix. And finally, when the Exchange is complete the local file is deleted.

So if you want to download files from a remote FTP server and store it as files then you need to route to a file endpoint such as:

```
from("ftp://someone@someserver.com?password=secret&localWorkDirectory=/tmp").to("file://in
```

```
box");
```

Optimization by renaming work file

The route above is ultra efficient as it avoids reading the entire file content into memory. It will download the remote file directly to a local file stream. The `java.io.File` handle is then used as the Exchange body. The file producer leverages this fact and can work directly on the work file `java.io.File` handle and perform a `java.io.File.rename` to the target filename. As Fuse Mediation Router knows it's a local work file, it can optimize and use a rename instead of a file copy, as the work file is meant to be deleted anyway.

Samples

In the sample below we set up Fuse Mediation Router to download all the reports from the FTP server once every hour (60 min) as BINARY content and store it as files on the local file system.

```
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // we use a delay of 60 minutes (eg. once pr. hour we poll the FTP server
            long delay = 60 * 60 * 1000L;

            // from the given FTP server we poll (= download) all the files
            // from the public/reports folder as BINARY types and store this as files
            // in a local directory. Fuse Mediation Router will use the filenames from the
 FTPServer

            // notice that the FTPConsumer properties must be prefixed with "consumer." in
 the URL
            // the delay parameter is from the FileConsumer component so we should use
consumer.delay as
           // the URI parameter name. The FTP Component is an extension of the File Compon
ent.
            from("ftp://tiger:scott@localhost/public/reports?binary=true&consumer.delay="
+ delay).
                to("file://target/test-reports");
        }
    };
}
```

And the route using Spring DSL:

```
  <route>
    <from uri="ftp://scott@localhost/public/reports?password=tiger&inary=true&elay=60000"/>
```

```
    <to uri="file://target/test-reports"/>
  </route>
```

Consuming a remote FTP server triggered by a route

The FTP consumer is built as a scheduled consumer to be used in the **from** route. However, if you want to start consuming from an FTP server triggered within a route, use a route like the following:

```
from("seda:start")
   // set the filename in FILE_NAME header so Fuse Mediation Router know the name of the
remote file to poll
   .setHeader(Exchange.FILE_NAME, header("myfile"))
   .pollEnrich("ftp://admin@localhost:21/getme?password=admin&binary=false")
   .to("mock:result");
```

Consuming a remote FTPS server (implicit SSL) and client authentication

```
from("ftps://admin@localhost:2222/public/camel?password=admin&securityProtocol=SSL&isImpli
cit=true
      &ftpClient.keyStore.file=./src/test/resources/server.jks
      &ftpClient.keyStore.password=password&ftpClient.keyStore.keyPassword=password")
  .to("bean:foo");
```

Consuming a remote FTPS server (explicit TLS) and a custom trust store configuration

```
from("ftps://admin@localhost:2222/public/camel?password=admin&ftpClient.trust
Store.file=./src/test/resources/server.jks&ftpClient.trustStore.password=password")
  .to("bean:foo");
```

Filter using org.apache.camel.component.file.GenericFileFilter

Fuse Mediation Router supports pluggable filtering strategies. You define a filter strategy by implementing the `org.apache.camel.component.file.GenericFileFilter` interface in Java. You can then configure the endpoint with the filter to skip certain files.

In the following sample we define a filter that only accepts files whose filename starts with `report`.

```
public class MyFileFilter implements GenericFileFilter {
```

```
    public boolean accept(GenericFile file) {
        // we only want report files
        return file.getFileName().startsWith("report");
    }
}
```

And then we can configure our route using the **filter** attribute to reference our filter (using # notation) that we have defined in the spring XML file:

```
   <!-- define our sorter as a plain spring bean -->
   <bean id="myFilter" class="com.mycompany.MyFileFilter"/>

  <route>
    <from uri="ftp://someuser@someftpserver.com?password=secret&filter=#myFilter"/>
    <to uri="bean:processInbox"/>
  </route>
```

Filtering using ANT path matcher

The ANT path matcher is a filter that is shipped out-of-the-box in the **camel-spring** jar. So you need to depend on **camel-spring** if you are using Maven. The reason is that we leverage Spring's AntPathMatcher[6] to do the actual matching.

The file paths are matched with the following rules:

- ? matches one character
- * matches zero or more characters
- ** matches zero or more directories in a path

The sample below demonstrates how to use it:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <template id="camelTemplate"/>

    <!-- use myFilter as filter to allow setting ANT paths for which files to scan for -->

   <endpoint id="myFTPEndpoint" uri="ftp://admin@localhost:20123/antpath?password=admin&ecurs
ive=true&elay=10000&nitialDelay=2000&ilter=#myAntFilter"/>

    <route>
        <from ref="myFTPEndpoint"/>
        <to uri="mock:result"/>
    </route>
```

[6] http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/util/AntPathMatcher.html

```
</camelContext>

<!-- we use the AntPathMatcherRemoteFileFilter to use ant paths for includes and exlucde -
->
<bean id="myAntFilter" class="org.apache.camel.component.file.AntPathMatcherGenericFileFil
ter">
    <!-- include and file in the subfolder that has day in the name -->
    <property name="includes" value="**/subfolder/**/*day*"/>
    <!-- exclude all files with bad in name or .xml files. Use comma to seperate multiple
excludes -->
    <property name="excludes" value="**/*bad*,**/*.xml"/>
</bean>
```

Debug logging

This component has log level **TRACE** that can be helpful if you have problems.

Chapter 29. GAE

Introduction to the GAE Components

Fuse Mediation Router Components for Google App Engine

Tutorials

- A good starting point for using Fuse Mediation Router on GAE is the Tutorial for Camel on Google App Engine[1]
- The OAuth tutorial[2] demonstrates how to implement OAuth[3] in web applications.

The Fuse Mediation Router components for Google App Engine[4] (GAE) are part of the `camel-gae` project and provide connectivity to GAE's cloud computing services[5]. They make the GAE cloud computing environment accessible to applications via Fuse Mediation Router interfaces. Following this pattern for other cloud computing environments could make it easier to port Fuse Mediation Router applications from one cloud computing provider to another. The following table lists the cloud computing services provided by Google App Engine and the supporting Fuse Mediation Router components. The documentation of each component can be found by following the link in the *Camel Component* column.

GAE service	Camel component	Component description
URL fetch service[6]	ghttp on page 193	Provides connectivity to the GAE URL fetch service but can also be used to receive messages from servlets.
Task queueing service[7]	gtask on page 205	Supports asynchronous message processing on GAE by using the task queueing service as message queue.
Mail service[8]	gmail on page 201	Supports sending of emails via the GAE mail service. Receiving mails is not supported yet but will be added later.
Memcache service[9]		Not supported yet.
XMPP service[10]		Not supported yet.

[1] Tutorial for Camel on Google App Engine
[2] Tutorial-OAuth
[3] http://oauth.net/
[4] http://code.google.com/appengine/
[5] http://code.google.com/appengine/docs/java/apis.html
[6] http://code.google.com/appengine/docs/java/urlfetch/
[7] http://code.google.com/appengine/docs/java/taskqueue/
[8] http://code.google.com/appengine/docs/java/mail/
[9] http://code.google.com/appengine/docs/java/memcache/
[10] http://code.google.com/appengine/docs/java/xmpp/

Images service[11]		Not supported yet.
Datastore service[12]		Not supported yet.
Accounts service[13]	gauth on page 184 glogin on page 197	These components interact with the Google Accounts API for authentication and authorization. Google Accounts is not specific to Google App Engine but is often used by GAE applications for implementing security. The gauth on page 184 component is used by web applications to implement a Google-specific OAuth[14] consumer. This component can also be used to OAuth-enable non-GAE web applications. The glogin on page 197 component is used by Java clients (outside GAE) for programmatic login to GAE applications. For instructions how to protect GAE applications against unauthorized access refer to the Security for Fuse Mediation Router GAE applications on page 203 page.

Camel context

Setting up a `SpringCamelContext` on Google App Engine differs between Camel 2.1 and higher versions. The problem is that usage of the Camel-specific Spring configuration XML schema from the `http://camel.apache.org/schema/spring` namespace requires JAXB and Camel 2.1 depends on a Google App Engine SDK version that doesn't support JAXB yet. This limitation has been removed since Camel 2.2.

JMX must be disabled in any case because the `javax.management` package isn't on the App Engine JRE whitelist.

Fuse Mediation Router 2.1

`camel-gae` 2.1 comes with the following `CamelContext` implementations.

- `org.apache.camel.component.gae.context.GaeDefaultCamelContext` (extends `org.apache.camel.impl.DefaultCamelContext`)
- `org.apache.camel.component.gae.context.GaeSpringCamelContext` (extends `org.apache.camel.spring.SpringCamelContext`)

Both disable JMX before startup. The `GaeSpringCamelContext` additionally provides setter methods adding route builders as shown in the next example.

[11] http://code.google.com/appengine/docs/java/images/
[12] http://code.google.com/appengine/docs/java/datastore/
[13] http://code.google.com/apis/accounts/
[14] http://code.google.com/apis/accounts/docs/OAuth.html

appctx.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="camelContext"
        class="org.apache.camel.component.gae.context.GaeSpringCamelContext">
        <property name="routeBuilder" ref="myRouteBuilder" />
    </bean>

    <bean id="myRouteBuilder"
        class="org.example.MyRouteBuilder">
    </bean>

</beans>
```

Alternatively, use the `routeBuilders` property of the `GaeSpringCamelContext` for setting a list of route builders. Using this approach, a `SpringCamelContext` can be configured on GAE without the need for JAXB.

Fuse Mediation Router 2.2

With Camel 2.2 or higher, applications can use the `http://camel.apache.org/schema/spring` namespace for configuring a `SpringCamelContext` but still need to disable JMX. Here's an example.

appctx.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

    <camel:camelContext id="camelContext">
        <camel:jmxAgent id="agent" disabled="true" />
        <camel:routeBuilder ref="myRouteBuilder"/>
    </camel:camelContext>

    <bean id="myRouteBuilder"
        class="org.example.MyRouteBuilder">
    </bean>

</beans>
```

The web.xml

Running Fuse Mediation Router on GAE requires usage of the `CamelHttpTransportServlet` from `camel-servlet`. The following example shows how to configure this servlet together with a Spring application context XML file.

web.xml

```
<web-app
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="
http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">

    <servlet>
        <servlet-name>CamelServlet</servlet-name>
       <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-
class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>appctx.xml</param-value>
        </init-param>
    </servlet>

    <!--
        Mapping used for external requests
     -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/camel/*</url-pattern>
    </servlet-mapping>

    <!--
        Mapping used for web hooks accessed by task queueing service.
     -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/worker/*</url-pattern>
    </servlet-mapping>

</web-app>
```

The location of the Spring application context XML file is given by the `contextConfigLocation` init parameter. The `appctx.xml` file must be on the classpath. The servlet mapping makes the Fuse Mediation Router application accessible under `http://<appname>.appspot.com/camel/...` when deployed to Google App Engine where `<appname>` must be replaced by a real GAE application name. The second servlet mapping is

used internally by the task queueing service for background processing via web hooks[15]. This mapping is relevant for the gtask on page 205 component and is explained there in more detail.

[15] http://www.webhooks.org/

gauth

gauth Component

Available in Fuse Mediation Router 2.3

The `gauth` component is used by web applications to implement a Google-specific OAuth[16] consumer. It will be later extended to support other OAuth[17] providers as well. Although this component belongs to the Camel Components for Google App Engine on page 177 (GAE), it can also be used to OAuth-enable non-GAE web applications. For a detailed description of Google's OAuth implementation refer to the Google OAuth API reference[18].

URI format

```
gauth://name[?options]
```

The endpoint `name` can be either `authorize` or `upgrade`. An `authorize` endpoint is used to obtain an unauthorized request token from Google and to redirect the user to the authorization page. The `upgrade` endpoint is used to process OAuth callbacks from Google and to upgrade an authorized request token to a long-lived access token. Refer to the usage section for an example.

Options

Name	Default Value	Required	Description
`callback`	`null`	true (can alternatively be set via `GAuthAuthorizeBinding.GAUTH_CALLBACK` message header)	URL where to redirect the granted or denied access.
`scope`	`null`	true (can alternatively be set via `GAuthAuthorizeBinding.GAUTH_SCOPE` message header)	URL identifying the service Scopes are defined by ea see the service's documen value. To specify more tha each one separated with a `http://www.google.com/` [19].
`consumerKey`	`null`	true (can alternatively be set on component-level).	Domain identifying the we is the domain used when application with Google. E

[16] http://code.google.com/apis/accounts/docs/OAuth.html
[17] http://oauth.net/
[18] http://code.google.com/apis/accounts/docs/OAuth_ref.html
[19] http://www.google.com/calendar/feeds/

			`camelcloud.appspot.` non-registered applicat
`consumerSecret`	`null`	one of `consumerSecret` or `keyLoaderRef` is required (can alternatively be set on component-level).	Consumer secret of the consumer secret is ger registering the applicat needed if the HMAC-S shall be used. For a nor use `anonymous`.
`keyLoaderRef`	`null`	one of `consumerSecret` or `keyLoaderRef` is required (can be alternatively set on component-level)	Reference to a private registry. Part of `camel-` loaders: `GAuthPk8Load` key from a PKCS#8 file to load a private key fr is needed if the RSA-S shall be used. These c the `org.apache.camel` package.
`authorizeBindingRef`	Reference to `GAuthAuthorizeBinding`	false	Reference to a `OutboundBinding<GAu` `GoogleOAuthParamete` `GoogleOAuthParamete` customizing how an `Ex` `GoogleOAuthParamete` for teh authorization ph won't change the defau
`upgradeBindingRef`	Reference to `GAuthAuthorizeBinding`	false	Reference to a `OutboundBinding<GAu` `GoogleOAuthParamete` `GoogleOAuthParamete` customizing how an `Ex` `GoogleOAuthParamete` for teh token upgrade ph won't change the defau

Message headers

Name	Type	Endpoint	Message	Description
`GAuthAuthorizeBinding.GAUTH_CALLBACK`	`String`	`gauth:authorize`	in	Overrides the

				`callback` option.
`GAuthAuthorizeBinding.GAUTH_SCOPE`	`String`	`gauth:authorize`	in	Overrides the `scope` option.
`GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN`	`String`	`gauth:upgrade`	out	Contains the long-lived access token. This token should be stored by the applications in context of a user.
`GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN_SECRET`	`String`	`gauth:upgrade`	out	Contains the access token secret. This token secret should be stored by the applications in context of a user.

Message body

The `gauth` component doesn't read or write message bodies.

Component configuration

Some endpoint options such as `consumerKey`, `consumerSecret` or `keyLoader` are usually set to the same values on `gauth:authorize` and `gauth:upgrade` endpoints. The `gauth` component allows to configure them on component-level. These settings are then inherited by `gauth` endpoints and need not be set redundantly in the endpoint URIs. Here are some configuration examples.

component configuration for a registered web application using the HMAC-SHA1 signature method

```
<bean id="gauth" class="org.apache.camel.component.gae.auth.GAuthComponent">
    <property name="consumerKey" value="example.appspot.com" />
    <property name="consumerSecret" value="QAtA...HfQ" />
</bean>
```

component configuration for an unregistered web application using the HMAC-SHA1 signature method

```
<bean id="gauth" class="org.apache.camel.component.gae.auth.GAuthComponent">
    <!-- Google will display a warning message on the authorization page -->
    <property name="consumerKey" value="anonymous" />
    <property name="consumerSecret" value="anonymous" />
</bean>
```

component configuration for a registered web application using the RSA-SHA1 signature method

```
<bean id="gauth" class="org.apache.camel.component.gae.auth.GAuthComponent">
    <property name="consumerKey" value="ipfcloud.appspot.com" />
    <property name="keyLoader" ref="jksLoader" />
    <!--<property name="keyLoader" ref="pk8Loader" />-->
</bean>

<!-- Loads the private key from a Java key store -->
<bean id="jksLoader"
    class="org.apache.camel.component.gae.auth.GAuthJksLoader">
    <property name="keyStoreLocation" value="myKeytore.jks" />
    <property name="keyAlias" value="myKey" />
    <property name="keyPass" value="myKeyPassword" />
    <property name="storePass" value="myStorePassword" />
</bean>

<!-- Loads the private key from a PKCS#8 file -->
<bean id="pk8Loader"
    class="org.apache.camel.component.gae.auth.GAuthPk8Loader">
    <property name="keyStoreLocation" value="myKeyfile.pk8" />
</bean>
```

Usage

Here's the minimum setup for adding OAuth to a (non-GAE) web application. In the following example, it is assumed that the web application is running on `gauth.example.org`.

GAuthRouteBuilder.java

```
import java.net.URLEncoder;
import org.apache.camel.builder.RouteBuilder;

public class GAuthRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

       // Calback URL to redirect user from Google Authorization back to the web application

       String encodedCallback = URLEncoder.encode("https://gauth.example.org:8443/handler",
 "UTF-8");
        // Application will request for authorization to access a user's Google Calendar
        String encodedScope = URLEncoder.encode("http://www.google.com/calendar/feeds/",
"UTF-8");

        // Route 1: A GET request to http://gauth.example.org/authorize will trigger the
the OAuth
        // sequence of interactions. The gauth:authorize endpoint obtains an unauthorized
request
       // token from Google and then redirects the user (browser) to a Google authorization
 page.
        from("jetty:http://0.0.0.0:8080/authorize")
            .to("gauth:authorize?callback=" + encodedCallback + "&scope=" + encodedScope);


        // Route 2: Handle callback from Google. After the user granted access to Google
Calendar
       // Google redirects the user to https://gauth.example.org:8443/handler (see callback)
 along
       // with an authorized request token. The gauth:access endpoint exchanges the author
ized
        // request token against a long-lived access token.
        from("jetty:https://0.0.0.0:8443/handler")
            .to("gauth:upgrade")
            // The access token can be obtained from
            // exchange.getOut().getHeader(GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN)
            // The access token secret can be obtained from
            // exchange.getOut().getHeader(GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN_SECRET)
            .process(/* store the tokens in context of the current user ... */);
    }
```

```
}
```

The OAuth sequence is triggered by sending a GET request to `http://gauth.example.org/authorize`[20]. The user is then redirected to a Google authorization page. After having granted access on this page, Google redirects the user to the web application which handles the callback and finally obtains a long-lived access token from Google.

These two routes can perfectly co-exist with any other web application framework. The framework provides the basis for web application-specific functionality whereas the OAuth service provider integration is done with Fuse Mediation Router. The OAuth integration part could even use resources from an existing servlet container by using the `servlet` component instead of the `jetty` component.

What to do with the OAuth access token?

- Application should store the access token in context of the current user. If the user logs in next time, the access token can directly be loaded from the database, for example, without doing the *OAuth dance* again.
- The access token is then used to get access to Google services, such as a Google Calendar API, on behalf of the user. Java applications will most likely use the GData Java library[21] for that. See below for an example how to use the access token with the GData Java library to read a user's calendar feed.
- The user can revoke the access token at any time from his Google Accounts[22] page. In this case, access to the corresponding Google service will throw an authorization exception. The web application should remove the stored access token and redirect the user again to the Google authorization page for creating another one.

The above example relies on the following component configuration.

```
<bean id="gauth" class="org.apache.camel.component.gae.auth.GAuthComponent">
    <property name="consumerKey" value="anonymous" />
    <property name="consumerSecret" value="anonymous" />
</bean>
```

[20] http://gauth.example.org/authorize
[21] http://code.google.com/p/gdata-java-client/
[22] https://www.google.com/accounts

If you don't want that Google displays a warning message on the authorization page, you'll need to register[23] your web application and change the `consumerKey` and `consumerSecret` settings.

GAE example

To OAuth-enable a Google App Engine application, only some small changes in the route builder are required. Assuming the GAE application hostname is `camelcloud.appspot.com` a configuration might look as follows. Here, the ghttp on page 193 component is used to handle HTTP(S) requests instead of the `jetty` component.

GAuthRouteBuilder

```
import java.net.URLEncoder;
import org.apache.camel.builder.RouteBuilder;

public class TutorialRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

       String encodedCallback = URLEncoder.encode("https://camelcloud.appspot.com/handler",
 "UTF-8");
        String encodedScope = URLEncoder.encode("http://www.google.com/calendar/feeds/",
"UTF-8");

        from("ghttp:///authorize")
            .to("gauth:authorize?callback=" + encodedCallback + "&scope=" + encodedScope);


        from("ghttp:///handler")
            .to("gauth:upgrade")
            .process(/* store the tokens in context of the current user ... */);
    }

}
```

[23] http://code.google.com/apis/accounts/docs/RegistrationForWebAppsAuto.html

Access token usage

Here's an example how to use an access token to access a user's Google Calendar data with the GData Java library[24]. The example application writes the titles of the user's public and private calendars to `stdout`.

Access token usage

```
import com.google.gdata.client.authn.oauth.OAuthHmacSha1Signer;
import com.google.gdata.client.authn.oauth.OAuthParameters;
import com.google.gdata.client.calendar.CalendarService;
import com.google.gdata.data.calendar.CalendarEntry;
import com.google.gdata.data.calendar.CalendarFeed;

import java.net.URL;

public class AccessExample {

    public static void main(String... args) throws Exception {
        String accessToken = ...
        String accessTokenSecret = ...

        CalendarService myService = new CalendarService("exampleCo-exampleApp-1.0");
        OAuthParameters params = new OAuthParameters();
        params.setOAuthConsumerKey("anonymous");
        params.setOAuthConsumerSecret("anonymous");
        params.setOAuthToken(accessToken);
        params.setOAuthTokenSecret(accessTokenSecret);
        myService.setOAuthCredentials(params, new OAuthHmacSha1Signer());

        URL feedUrl = new URL("http://www.google.com/calendar/feeds/default/");
        CalendarFeed resultFeed = myService.getFeed(feedUrl, CalendarFeed.class);

        System.out.println("Your calendars:");
        System.out.println();

        for (int i = 0; i < resultFeed.getEntries().size(); i++) {
            CalendarEntry entry = resultFeed.getEntries().get(i);
            System.out.println(entry.getTitle().getPlainText());
        }
    }
}
```

[24] http://code.google.com/p/gdata-java-client/

ghttp

ghttp Component

The `ghttp` component contributes to the Camel Components for Google App Engine on page 177 (GAE). It provides connectivity to the GAE URL fetch service[25] but can also be used to receive messages from servlets (the only way to receive HTTP requests on GAE). This is achieved by extending the `Servlet` component on page 435. As a consequence, `ghttp` URI formats and options sets differ on the consumer-side (`from`) and producer-side (`to`).

URI format

Format	Context	Comment
`ghttp:///path[?options]`	Consumer	See also Servlet component on page 435
`ghttp://hostname[:port][/path][?options]` `ghttps://hostname[:port][/path][?options]`	Producer	See also Http component on page 229

Options

Name	Default Value	Context	Description
`bridgeEndpoint`	`true`	Producer	If set to `true` the `Exchange.HTTP_URI` header will be ignored. To override the default endpoint URI with the `Exchange.HTTP_URI` header set this option to `false`.
`throwExceptionOnFailure`	`true`	Producer	Throw a `org.apache.camel.component.gae.http` if the response code is >= 400. To disable throwing an exception set this option to `false`.
`inboundBindingRef`	reference to `GHttpBinding`	Consumer	Reference to an `InboundBinding<GHttpEndpoint, HttpServletRequest, HttpServletResponse>` in the Registry for customizing the binding of an `Exchange` to the Servlet API. The referenced binding is used as post-processor to `org.apache.camel.component.http.HttpBinding`.
`outboundBindingRef`	reference to `GHttpBinding`	Producer	Reference to an `OutboundBinding<GHttpEndpoint, HTTPRequest, HTTPResponse>` in the Registry for customizing the binding of an `Exchange` to the `URLFetchService`.

[25] http://code.google.com/appengine/docs/java/urlfetch/

On the consumer-side, all options of the Servlet component on page 435 are supported.

Message headers

On the producer side, the following headers of the Http component on page 229 are supported.

Name	Type	Description
`Exchange.CONTENT_TYPE`	`String`	The HTTP content type. Is set on both the `in` and `out` message to provide a content type, such as `text/html`.
`Exchange.CONTENT_ENCODING`	`String`	The HTTP content encoding. Is set on both the `in` and `out` message to provide a content encoding, such as `gzip`.
`Exchange.HTTP_METHOD`	`String`	The HTTP method to execute. One of `GET`, `POST`, `PUT` and `DELETE`. If not set, `POST` will be used if the message body is not `null`, `GET` otherwise.
`Exchange.HTTP_QUERY`	`String`	Overrides the query part of the endpoint URI or the the query part of `Exchange.HTTP_URI` (if defined). The query string must be in decoded form.
`Exchange.HTTP_URI`	`String`	Overrides the default endpoint URI if the `bridgeEndpoint` option is set to `false`. The URI string must be in decoded form.
`Exchange.RESPONSE_CODE`	`int`	The HTTP response code from URL fetch service responses.

On the consumer-side all headers of the Servlet component on page 435 component are supported.

Message body

On the producer side the `in` message body is converted to a `byte[]`. The `out` message body is made available as `InputStream`. If the reponse size exceeds 1 megabyte a ResponseTooLargeException[26] is thrown by the URL fetch service (see quotas and limits[27]).

Receiving messages

For receiving messages via the `ghttp` component, a `CamelHttpTransportServlet` must be configured and mapped in the application's `web.xml`. For example, to handle requests targeted at `http://<appname>.appspot.com/camel/*` or `http://localhost/camel/*` (when using a local development server) the following servlet mapping must be defined:

[26] http://code.google.com/appengine/docs/java/javadoc/
[27] http://code.google.com/appengine/docs/java/urlfetch/overview.html#Quotas_and_Limits

web.xml

```
    ...
    <servlet>
        <servlet-name>CamelServlet</servlet-name>
       <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-
class>
        ...
    </servlet>
    ...
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/camel/*</url-pattern>
    </servlet-mapping>
    ...
```

Endpoint URI path definitions are relative to this servlet mapping e.g. the route

```
from("ghttp:///greeting").transform().constant("Hello")
```

processes requests targeted at `http://<appname>.appspot.com/camel/greeting`. In this example, the request body is ignored and the response body is set to `Hello`. Requests targeted at `http://<appname>.appspot.com/camel/greeting/*` are not processed by default. This requires setting the option `matchOnUriPrefix` to `true`.

```
from("ghttp:///greeting?matchOnUriPrefix=true").transform().constant("Hello")
```

Sending messages

For sending resquests to external HTTP services the `ghttp` component uses the URL fetch service[28]. For example, the Fuse Mediation Router homepage can the retrieved with the following endpoint definition on the producer-side.

```
from(...)
...
.to("ghttp://camel.apache.org")
...
```

The HTTP method used depends on the `Exchange.HTTP_METHOD` message header or on the presence of an in-message body (`GET` if `null`, `POST` otherwise). Retrieving the Camel homepage via a GAE application is as simple as

[28] http://code.google.com/appengine/docs/java/urlfetch/

```
from("ghttp:///home")
.to("ghttp://camel.apache.org")
```

Sending a `GET` request to `http://<appname>.appspot.com/camel/home` returns the Camel homepage. HTTPS-based communication with external services can be enabled with the `ghttps` scheme.

```
from(...)
...
.to("ghttps://svn.apache.org/repos/asf/camel/trunk/")
...
```

Dependencies

Maven users will need to add the following dependency to their `pom.xml`.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-gae</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where `${camel-version}` must be replaced by the actual version of Fuse Mediation Router (2.1.0 or higher).

glogin

glogin Component

Available in Fuse Mediation Router 2.3 (or latest development snapshot[29]).

The `glogin` component is used by Fuse Mediation Router applications outside Google App Engine (GAE) for programmatic login to GAE applications. It is part of the Fuse Mediation Router Components for Google App Engine on page 177. Security-enabled GAE applications on page 203 normally redirect the user to a login page. After submitting username and password for authentication, the user is redirected back to the application. That works fine for applications where the client is a browser. For all other applications, the login process must be done programmatically. All the necessary steps[30] for programmatic login are implemented by the `glogin` component. These are

1. Get an authentication token from Google Accounts[31] via the ClientLogin API[32].

2. Get an authorization cookie from Google App Engine's login API.

The authorization cookie must then be send with subsequent HTTP requests to the GAE application. It expires after 24 hours and must then be renewed.

URI format

```
glogin://hostname[:port][?options]
```

The `hostname` is either the internet hostname of a GAE application (e.g. `camelcloud.appspot.com`) or the name of the host where the development server[33] is running (e.g. `localhost`). The `port` is only used when connecting to a development server (i.e. when `devMode=true`, see options) and defaults to `8080`.

Options

Name	Default Value	Required	Description
`clientName`	`apache-camel-2.x`	false	A client name with recommended (but not required) format `<organization>\-<appname>\-<version>`.

[29] https://svn.apache.org/repos/asf/camel/trunk/components/camel-gae/
[30] http://krasserm.blogspot.com/2010/01/accessing-security-enabled-google-app.html
[31] http://code.google.com/apis/accounts/
[32] http://code.google.com/apis/accounts/docs/AuthForInstalledApps.html
[33] http://code.google.com/appengine/docs/java/tools/devserver.html

`userName`	`null`	true (can alternatively be set via `GLoginBinding.GLOGIN_USER_NAME` message header)	Login username (an email address).
`password`	`null`	true (can alternatively be set via `GLoginBinding.GLOGIN_PASSWORD` message header)	Login password.
`devMode`	`false`	false	If set to `true` a login to a development server is attempted.
`devAdmin`	`false`	false	If set to `true` a login to a development server in admin role is attempted.

Message headers

Name	Type	Message	Description
`GLoginBinding.GLOGIN_HOST_NAME`	`String`	in	Overrides the hostname defined in the endpoint URI.
`GLoginBinding.GLOGIN_USER_NAME`	`String`	in	Overrides the `userName` option.
`GLoginBinding.GLOGIN_PASSWORD`	`String`	in	Overrides the `password` option.
`GLoginBinding.GLOGIN_TOKEN`	`String`	out	Contains the authentication token obtained from Google Accounts[34]. Login to a development server does not set this header.
`GLoginBinding.GLOGIN_COOKIE`	`String`	out	Contains the application-specific authorization cookie obtained from Google App Engine (or a development server).

Message body

The `glogin` component doesn't read or write message bodies.

Usage

The following JUnit test show an example how to login to a development server as well as to a deployed GAE application located at http://camelcloud.appspot.com.

[34] http://code.google.com/apis/accounts/

GLoginTest.java

```
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.ProducerTemplate;
import org.junit.Ignore;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import static org.apache.camel.component.gae.login.GLoginBinding.*;
import static org.junit.Assert.*;

public class GLoginTest {

    private ProducerTemplate template = ...

    @Test
    public void testDevLogin() {
        Exchange result = template.request("glogin://localhost:8888?userName=test@ex
ample.org&devMode=true", null);
        assertNotNull(result.getOut().getHeader(GLOGIN_COOKIE));
    }

    @Test
    public void testRemoteLogin() {
       Exchange result = template.request("glogin://camelcloud.appspot.com", new Processor()
 {
            public void process(Exchange exchange) throws Exception {
                exchange.getIn().setHeader(GLOGIN_USER_NAME, "replaceme@gmail.com");
                exchange.getIn().setHeader(GLOGIN_PASSWORD, "replaceme");
            }
        });
        assertNotNull(result.getOut().getHeader(GLOGIN_COOKIE));
    }

}
```

The resulting authorization cookie from login to a development server looks like

```
ahlogincookie=test@example.org:false:11223191102230730701;Path=/
```

The resulting authorization cookie from login to a deployed GAE application looks (shortened) like

```
ACSID=AJKiYcE...XxhH9P_jR_V3; expires=Sun, 07-Feb-2010 15:14:51 GMT; path=/
```

gmail

gmail Component

The `gmail` component contributes to the Camel Components for Google App Engine on page 177 (GAE). It supports sending of emails via the GAE mail service[35]. Receiving mails is not supported yet but will be added later. Currently, only Google accounts that are application administrators can send emails.

URI format

```
gmail://user@gmail.com[?options]
gmail://user@googlemail.com[?options]
```

Options

Name	Default Value	Context	Description
`to`	`null`	Producer	To-receiver of the email. This can be a single receiver or a comma-separated list of receivers.
`cc`	`null`	Producer	Cc-receiver of the email. This can be a single receiver or a comma-separated list of receivers.
`bcc`	`null`	Producer	Bcc-receiver of the email. This can be a single receiver or a comma-separated list of receivers.
`subject`	`null`	Producer	Subject of the email.
`outboundBindingRef`	reference to `GMailBinding`	Producer	Reference to an `OutboundBinding<GMailEndpoint, MailService.Message, void>` in the Registry for customizing the binding of an `Exchange` to the mail service.

Message headers

Name	Type	Context	Description
`GMailBinding.GMAIL_SUBJECT`	`String`	Producer	Subject of the email. Overrides `subject` endpoint option.
`GMailBinding.GMAIL_SENDER`	`String`	Producer	Sender of the email. Overrides sender definition in endpoint URI.
`GMailBinding.GMAIL_TO`	`String`	Producer	To-receiver(s) of the email. Overrides `to` endpoint option.

[35] http://code.google.com/appengine/docs/java/mail/

`GMailBinding.GMAIL_CC`	`String`	Producer	Cc-receiver(s) of the email. Overrides `cc` endpoint option.
`GMailBinding.GMAIL_BCC`	`String`	Producer	Bcc-receiver(s) of the email. Overrides `bcc` endpoint option.

Message body

On the producer side the `in` message body is converted to a `String`.

Usage

```
...
.setHeader(GMailBinding.GMAIL_SUBJECT, constant("Hello"))
.setHeader(GMailBinding.GMAIL_TO, constant("account2@somewhere.com"))
.to("gmail://account1@gmail.com");
```

Sends an email with subject `Hello` from `account1@gmail.com` to `account2@somewhere.com`. The mail message body is taken from the `in` message body. Please note that `account1@gmail.com` must be an administrator account for the current GAE application.

Dependencies

Maven users will need to add the following dependency to their `pom.xml`.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-gae</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where `$\{camel-version\}` must be replaced by the actual version of Fuse Mediation Router (2.1.0 or higher).

gsec

Security for Fuse Mediation Router GAE Applications

Securing GAE applications from unauthorized access is described in the Security and Authentication[36] section of the Google App Engine documentation. Authorization constraints are declared in the web.xml. This applies to Fuse Mediation Router applications as well. In the following example, the application is configured to only allow authenticated users (in any role) to access the application. Additionally, access to `/worker/\*` URLs masy only be done by users in the admin role. By default, web hook URLs installed by the gtask on page 205 component match the `/worker/\*` pattern and should not be accessed by normal users. With this authorization constraint, only the task queuing service (which is always in the admin role) is allowed to access the web hooks. For implementing custom, non-declarative authorization logic, Fuse Mediation Router GAE applications should use the Google Accounts Java API[37].

Example 29.1. web.xml with authorization constraint

```
<web-app
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="
http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">

    <servlet>
        <servlet-name>CamelServlet</servlet-name>
       <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-
class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>appctx.xml</param-value>
        </init-param>
    </servlet>

    <!--
        Mapping used for external requests
     -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/camel/*</url-pattern>
    </servlet-mapping>

    <!--
```

[36] http://code.google.com/appengine/docs/java/config/webxml.html#Security_and_Authentication
[37] http://code.google.com/appengine/docs/java/users/overview.html

```
        Mapping used for web hooks accessed by task queueing service.
     -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/worker/*</url-pattern>
    </servlet-mapping>

    <!--
        By default allow any user who is logged in to access the whole
        application.
     -->
    <security-constraint>
        <web-resource-collection>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>*</role-name>
        </auth-constraint>
    </security-constraint>

    <!--
        Allow only admin users to access /worker/* URLs e.g. to prevent
        normal user to access gtask web hooks.
     -->
    <security-constraint>
        <web-resource-collection>
            <url-pattern>/worker/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
        </auth-constraint>
    </security-constraint>

</web-app>
```

gtask

gtask Component

The `gtask` component contributes to the Camel Components for Google App Engine on page 177 (GAE). It supports asynchronous message processing on GAE by using the task queueing service[38] as message queue. For adding messages to a queue it uses the task queue API. For receiving messages from a queue it installs an HTTP callback handler. The handler is called by an HTTP POST callback (a web hook[39]) initiated by the task queueing service. Whenever a new task is added to a queue a callback will be sent. The `gtask` component abstracts from these details and supports endpoint URIs that make message queueing on GAE as easy as message queueing with JMS on page 293 or SEDA on page 429.

URI format

```
gtask://queue-name
```

Options

Name	Default Value	Context	Description
`workerRoot`	`worker`	Producer	The servlet mapping for callback handlers. By default, this component requires a callback servlet mapping of `/worker/*`. If another servlet mapping is used e.g. `/myworker/*` it must be set as option on the producer side: `to("gtask:myqueue?workerRoot=myworker")`.
`inboundBindingRef`	reference to `GTaskBinding`	Consumer	Reference to an `InboundBinding<GTaskEndpoint, HttpServletRequest, HttpServletResponse>` in the Registry for customizing the binding of an `Exchange` to the Servlet API. The referenced binding is used as post-processor to `org.apache.camel.component.http.HttpBinding`.
`outboundBindingRef`	reference to `GTaskBinding`	Producer	Reference to an `OutboundBinding<GTaskEndpoint, TaskOptions, void>` in the Registry for customizing the binding of an `Exchange` to the task queueing service.

On the consumer-side, all options of the Servlet component on page 435 are supported.

[38] http://code.google.com/appengine/docs/java/taskqueue/
[39] http://www.webhooks.org/

Message headers

On the consumer-side all headers of the Servlet component on page 435 component are supported plus the following.

Name	Type	Context	Description
`GTaskBinding.GTASK_QUEUE_NAME`	`String`	Consumer	Name of the task queue.
`GTaskBinding.GTASK_TASK_NAME`	`String`	Consumer	Name of the task (generated value).
`GTaskBinding.GTASK_RETRY_COUNT`	`int`	Consumer	Number of callback retries.

Message body

On the producer side the `in` message body is converted to a `byte[]` and is POSTed to the callback handler as content-type `application/octet-stream`.

Usage

Setting up tasks queues is an administrative task on Google App Engine. Only one queue is pre-configured and can be referenced by name out-of-the-box: the `default` queue. This queue will be used in the following examples. Please note that when using task queues on the local development server, tasks must be executed manually from the developer console[40].

Default queue

```
...
.to(gtask:default) // add message to default queue

from(gtask:default) // receive message from default queue (via a web hook)
...
```

This example requires the following servlet mapping.

[40] http://code.google.com/appengine/docs/java/taskqueue/overview.html#Task_Queues_and_the_Development_Server

web.xml

```
    ...
    <servlet>
        <servlet-name>CamelServlet</servlet-name>
       <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-
class>
        ...
    </servlet>
    ...
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/worker/*</url-pattern>
    </servlet-mapping>
    ...
```

Dependencies

Maven users will need to add the following dependency to their `pom.xml`.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-gae</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where `${camel-version}` must be replaced by the actual version of Fuse Mediation Router (2.1.0 or higher).

Chapter 30. HawtDB

HawtDB

Available as of Fuse Mediation Router 2.3

HawtDB[1] is a very lightweight and embedable key value database. It allows together with Fuse Mediation Router to provide persistent support for various Fuse Mediation Router features such as Aggregator.

Current features it provides:

- HawtDBAggregationRepository

Using HawtDBAggregationRepository

`HawtDBAggregationRepository` is an `AggregationRepository` which on the fly persists the aggregated messages. This ensures that you will not loose messages, as the default aggregator will use an in memory only `AggregationRepository`.

It has the following options:

Option	Type	Description
`repositoryName`	String	A mandatory repository name. Allows you to use a shared `HawtDBFile` for multiple repositories.
`persistentFileName`	String	Filename for the persistent storage. If no file exists on startup a new file is created.
`bufferSize`	int	The size of the memory segment buffer which is mapped to the file store. By default its 8mb. The value is in bytes.
`sync`	boolean	Whether or not the `HawtDBFile` should sync on write or not. Default is `true`. By sync on write ensures that its always waiting for all writes to be spooled to disk and thus will not loose updates. If you disable this option, then HawtDB will auto sync when it has batched up a number of writes.
`pageSize`	short	The size of memory pages. By default its 512 bytes. The value is in bytes.
`hawtDBFile`	HawtDBFile	Use an existing configured `org.apache.camel.component.hawtdb.HawtDBFile` instance.

[1] http://hawtdb.fusesource.org/

`returnOldExchange`	boolean	Whether the get operation should return the old existing Exchange if any existed. By default this option is `false` to optimize as we do not need the old exchange when aggregating.
`useRecovery`	boolean	Whether or not recovery is enabled. This option is by default `true`. When enabled the Fuse Mediation Router Aggregator automatic recover failed aggregated exchange and have them resubmitted.
`recoveryInterval`	long	If recovery is enabled then a background task is run every x'th time to scan for failed exchanges to recover and resubmit. By default this interval is 5000 millis.
`maximumRedeliveries`	int	Allows you to limit the maximum number of redelivery attempts for a recovered exchange. If enabled then the Exchange will be moved to the dead letter channel if all redelivery attempts failed. By default this option is disabled. If this option is used then the `deadLetterUri` option must also be provided.
`deadLetterUri`	String	An endpoint uri for a Dead Letter Channel where exhausted recovered Exchanges will be moved. If this option is used then the `maximumRedeliveries` option must also be provided.

The `repositoryName` option must be provided. Then either the `persistentFileName` or the `hawtDBFile` must be provided.

What is preserved when persisting

`HawtDBAggregationRepository` will only preserve any `Serializable` compatible data types. If a data type is not such a type its dropped and a `WARN` is logged. And it only persists the `Message` body and the `Message` headers. The `Exchange` properties are **not** persisted.

Recovery

The `HawtDBAggregationRepository` will by default recover any failed Exchange. It does this by having a background tasks that scans for failed Exchanges in the persistent store. You can use the `checkInterval` option to set how often this task runs. The recovery works as transactional which ensures that Fuse Mediation Router will try to recover and redeliver the failed Exchange. Any Exchange which was found to be recovered will be restored from the persistent store and resubmitted and send out again.

The following headers is set when an Exchange is being recovered/redelivered:

Header	Type	Description
`Exchange.REDELIVERED`	Boolean	Is set to true to indicate the Exchange is being redelivered.
`Exchange.REDELIVERY_COUNTER`	Integer	The redelivery attempt, starting from 1.

Only when an Exchange has been successfully processed it will be marked as complete which happens when the `confirm` method is invoked on the `AggregationRepository`. This means if the same Exchange fails again it will be kept retried until it success.

You can use option `maximumRedeliveries` to limit the maximum number of redelivery attempts for a given recovered Exchange. You must also set the `deadLetterUri` option so Fuse Mediation Router knows where to send the Exchange when the `maximumRedeliveries` was hit.

You can see some examples in the unit tests of camel-hawtdb, for example this test[2].

Using HawtDBAggregationRepository in Java DSL

In this example we want to persist aggregated messages in the `target/data/hawtdb.dat` file.

```
public void configure() throws Exception {
    // create the hawtdb repo
    HawtDBAggregationRepository repo = new HawtDBAggregationRepository("repo1", "tar
get/data/hawtdb.dat");

    // here is the Camel route where we aggregate
    from("direct:start")
        .aggregate(header("id"), new MyAggregationStrategy())
            // use our created hawtdb repo as aggregation repository
            .completionSize(5).aggregationRepository(repo)
            .to("mock:aggregated");
}
```

Using HawtDBAggregationRepository in Spring XML

The same example but using Spring XML instead:

```
<!-- a persistent aggregation repository using camel-hawtdb -->
<bean id="repo" class="org.apache.camel.component.hawtdb.HawtDBAggregationRepository">
    <!-- store the repo in the hawtdb.dat file -->
    <property name="persistentFileName" value="target/data/hawtdb.dat"/>
    <!-- and use repo2 as the repository name -->
    <property name="repositoryName" value="repo2"/>
</bean>

<!-- aggregate the messages using this strategy -->
<bean id="myAggregatorStrategy" class="org.apache.camel.component.hawtdb.HawtDBSpringAggreg
ateTest$MyAggregationStrategy"/>

<!-- this is the camel routes -->
```

[2] https://svn.apache.org/repos/asf/camel/trunk/components/camel-hawtdb/src/test/java/org/apache/camel/component/hawtdb/HawtDBAggregateRecoverTest.java

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

    <route>
        <from uri="direct:start"/>
        <!-- aggregate using our strategy and hawtdb repo, and complete when we have 5
messages aggregated -->
        <aggregate strategyRef="myAggregatorStrategy" aggregationRepositoryRef="repo" com
pletionSize="5">
            <!-- correlate by header with the key id -->
            <correlationExpression><header>id</header></correlationExpression>
            <!-- send aggregated messages to the mock endpoint -->
            <to uri="mock:aggregated"/>
        </aggregate>
    </route>

</camelContext>
```

Dependencies

To use HawtDB on page 209 in your Fuse Mediation Router routes you need to add the a dependency on **camel-hawtdb**.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions[3]).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hawtdb</artifactId>
  <version>2.3.0</version>
</dependency>
```

See Also:

- Aggregator
- JDBC-AggregationRepository on page 271
- Components on page 3

[3] Download

Chapter 31. HDFS

HDFS Component

The **hdfs** component enables you to read and write messages from/to an HDFS file system. HDFS is the distributed file system at the heart of Hadoop[1]. It can only be built using JDK1.6 or later because this is a strict requirement for Hadoop itself. This component is hosted at http://github.com/dgreco/camel-hdfs. We decided to put it temporarily on this github[2] because currently Camel is being built and tested using JDK1.5 and for this reason we couldn't put that component into the Camel official distribution. Hopefully, as soon Camel will allow to use JDK1.6 for building and testing we will put it into the trunk. This component is developed and tested using the latest Camel snapshot, but it should work seamlessly with the latest Camel GA version (at the time of writing 2.1.0)

URI format

```
hdfs://hostname[:port][/path][?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...` The path is treated in the following way:

1. as a consumer, if it's a file, it just reads the file, otherwise if it represents a directory it scans all the file under the path satisfying the configured pattern. All the files under that directory must be of the same type.

2. as a producer, if at least one split strategy is defined, the path is considered a directory and under that directory the producer creates a different file per split named seg0, seg1, seg2, etc.

Options

Name	Default Value	Description
`overwrite`	`true`	The file can be overwritten
`bufferSize`	`4096`	The buffer size used by HDFS
`replication`	`3`	The HDFS replication factor
`blockSize`	`67108864`	The size of the HDFS blocks
`fileType`	`NORMAL_FILE`	It can be SEQUENCE_FILE, MAP_FILE, ARRAY_FILE, or BLOOMMAP_FILE, see Hadoop
`fileSystemType`	`HDFS`	It can be LOCAL for local filesystem

[1] http://hadoop.apache.org
[2] http://www.github.com

`keyType`	`NULL`	The type for the key in case of sequence or map files. See below.
`valueType`	`TEXT`	The type for the key in case of sequence or map files. See below.
`splitStrategy`		A string describing the strategy on how to split the file based on different criteria. See below.
`openedSuffix`	`opened`	When a file is opened for reading/ writing the file is renamed with this suffix to avoid to read it during the writing phase.
`readSuffix`	`read`	Once the file has been read is renamed with this suffix to avoid to read it again.
`initialDelay`	`0`	For the consumer, how much to wait (milliseconds) before to start scanning the directory.
`delay`	`0`	The interval (milliseconds) between the directory scans.
`pattern`	`*`	The pattern used for scanning the directory
`chunkSize`	`4096`	When reading a normal file, this is split into chunks producing a message per chunk

KeyType and ValueType

- NULL it means that the key or the value is absent
- BYTE for writing a byte, the java Byte class is mapped into a BYTE
- BYTES for writing a sequence of bytes. It maps the java ByteBuffer class
- INT for writing java integer
- FLOAT for writing java float
- LONG for writing java long
- DOUBLE for writing java double
- TEXT for writing java strings

BYTES is also used with everything else, for example, in Camel a file is sent around as an InputStream, int this case is written in a sequence file or a map file as a sequence of bytes.

Splitting Strategy

In the current version of Hadoop (0.20.1) opening a file in append mode is disabled since it's not enough reliable. So, for the moment, it's only possible to create new files. The Camel HDFS endpoint tries to solve this problem in this way:

- If the split strategy option has been defined, the actual file name will become a directory name and a <file name>/seg0 will be initially created.
- Every time a splitting condition is met a new file is created with name <original file name>/segN where N is 1, 2, 3, etc.The splitStrategy option is defined as a string with the following syntax:splitStrategy=<ST>:<value>,<ST>:<value>,*

where <ST> can be: BYTES a new file is created, and the old is closed when the number of written bytes is more than <value> MESSAGES a new file is created, and the old is closed when the number of written messages is more than <value> IDLE a new file is created, and the old is closed when no writing happened in the last <value> milliseconds

for example: hdfs://localhost/tmp/simple-file?splitStrategy=IDLE:1000,BYTES:5 it means: a new file is created either when it has been idle for more than 1 second or if more than 5 bytes have been written. So, running hadoop fs *ls /tmp/simple*file you'll find the following files seg0, seg1, seg2, etc

Chapter 32. Hibernate

Hibernate Component

The **hibernate:** component allows you to work with databases using Hibernate as the object relational mapping technology to map POJOs to database tables. The **camel-hibernate** library is provided by the Camel Extra[1] project which hosts all GPL related components for Fuse Mediation Router.

Sending to the endpoint

Sending POJOs to the hibernate endpoint inserts entities into the database. The body of the message is assumed to be an entity bean that you have mapped to a relational table using the hibernate `.hbm.xml` files.

If the body does not contain an entity bean, use a Message Translator in front of the endpoint to perform the necessary conversion first.

Consuming from the endpoint

Consuming messages removes (or updates) entities in the database. This allows you to use a database table as a logical queue; consumers take messages from the queue and then delete/update them to logically remove them from the queue.

If you do not wish to delete the entity when it has been processed, you can specify `consumeDelete=false` on the URI. This will result in the entity being processed each poll.

If you would rather perform some update on the entity to mark it as processed (such as to exclude it from a future query) then you can annotate a method with @Consumed[2] which will be invoked on your entity bean when the entity bean is consumed.

URI format

```
hibernate:[entityClassName][?options]
```

For sending to the endpoint, the **entityClassName** is optional. If specified it is used to help use the Type Conversion[3] to ensure the body is of the correct type.

For consuming, the **entityClassName** is mandatory.

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

[1] http://code.google.com/p/camel-extra/
[2] http://activemq.apache.org/camel/maven/camel-hibernate/apidocs/org/apache/camel/component/hibernate/Consumed.html
[3] Type Conversion

Options

Name	Default Value	Description
`entityType`	*entityClassName*	Is the provided *entityClassName* from the URI.
`consumeDelete`	`true`	Option for HibernateConsumer only. Specifies whether or not the entity is deleted after it is consumed.
`consumeLockEntity`	`true`	Option for HibernateConsumer only. Specifies whether or not to use exclusive locking of each entity while processing the results from the pooling.
`flushOnSend`	`true`	Option for HibernateProducer only. Flushes the EntityManager[4] after the entity bean has been persisted.
`maximumResults`	`-1`	Option for HibernateConsumer only. Set the maximum number of results to retrieve on the Query[5].
`consumer.delay`	`500`	Option for HibernateConsumer only. Delay in millis between each poll.
`consumer.initialDelay`	`1000`	Option for HibernateConsumer only. Millis before polling starts.
`consumer.userFixedDelay`	`false`	Option for HibernateConsumer only. Set to `true` to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService[6] in JDK for details.

[4] http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html
[5] http://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html
[6] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html

Chapter 33. HL7

HL7 Component

The **hl7** component is used for working with the HL7 MLLP protocol and the HL7 model[1] using the HAPI library[2].

This component supports the following:

- HL7 MLLP codec for Mina[3].
- Agnostic data format using either plain String objects or HAPI HL7 model objects.
- Type Converter from/to HAPI and String.
- HL7 DataFormat using HAPI library.
- Even more easy-of-use as it is integrated well with the camel-mina component.

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

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-hl7</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

HL7 MLLP protocol

HL7 is often used with the HL7 MLLP protocol that is a text based TCP socket based protocol. This component ships with a Mina Codec that conforms to the MLLP protocol so you can easily expose a HL7 listener that accepts HL7 requests over the TCP transport.

To expose a HL7 listener service we reuse the existing camel-mina component where we just use the `HL7MLLPCodec` as codec.

The HL7 MLLP codec has the following options:

[1] http://www.hl7.org/
[2] http://hl7api.sourceforge.net
[3] http://mina.apache.org/

Name	Default Value	Description
`startByte`	`0x0b`	The start byte spanning the HL7 payload. Is the HL7 default value of `0x0b` (11 decimal).
`endByte1`	`0x1c`	The first end byte spanning the HL7 payload. Is the HL7 default value of `0x1c` (28 decimal).
`endByte2`	`0x0d`	The 2nd end byte spanning the HL7 payload. Is the HL7 default value of `0x0d` (13 decimal).
`charset`	JVM Default	The encoding (is a charset name[4]) to use for the codec. If not provided, Fuse Mediation Router will use the JVM default Charset[5].
`convertLFtoCR`	`true`	Will convert `\n` to `\r` (`0x0d`, 13 decimal) as HL7 usually uses `\r` as segment terminators. The HAPI library requires the use of `\r`.
`validate`	`true`	**Fuse Mediation Router 2.0:** Whether HAPI Parser should validate or not.

Exposing a HL7 listener

In our Spring XML file, we configure an endpoint to listen for HL7 requests using TCP:

```
      <endpoint id="hl7listener" uri="mina:tcp://localhost:8888?sync=true&codec=#hl7codec"/>
```

Notice we configure it to use camel-mina with TCP on the `localhost` on port `8888`. We use **sync=true** to indicate that this listener is synchronous and therefore will return a HL7 response to the caller. Then we set up mina to use our HL7 codec with **codec=#hl7codec**. Notice that `hl7codec` is just a Spring bean ID, so we could have named it `mygreatcodecforhl7` or whatever. The codec can also be set up in the Spring XML file:

```
    <bean id="hl7codec" class="org.apache.camel.component.hl7.HL7MLLPCodec">
        <property name="charset" value="iso-8859-1"/>
    </bean>
```

And here we configure the charset encoding to use, and `iso-8859-1` is commonly used.

The endpoint **hl7listener** can then be used in a route as a consumer, as this java DSL example illustrates:

```
    from("hl7listener").to("patientLookupService");
```

This is a very simple route that will listen for HL7 and route it to a service named **patientLookupService** that is also a Spring bean ID we have configured in the Spring XML as:

```
   <bean id="patientLookupService" class="com.mycompany.healtcare.service.PatientLookupSer
vice"/>
```

[4] http://java.sun.com/j2se/1.5.0/docs/api/java/nio/charset/Charset.html
[5] http://java.sun.com/j2se/1.5.0/docs/api/java/nio/charset/Charset.html#defaultCharset()

And another powerful feature of Fuse Mediation Router is that we can have our busines logic in POJO classes that are not at all tied to Fuse Mediation Router as shown here:

```
public class PatientLookupService {
    public Message lookupPatient(Message input) throws HL7Exception {
        QRD qrd = (QRD)input.get("QRD");
        String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue();

       // find patient data based on the patient id and create a HL7 model object with the
 response
        Message response = ... create and set response data
        return response
    }
```

Notice that this class is just using imports from the HAPI library and **none** from Fuse Mediation Router.

HL7 Model using java.lang.String

The HL7MLLP codec uses plain `String` as data format. And Fuse Mediation Router uses Type Converter to convert from/to strings to the HAPI HL7 model objects. However, you can use the plain `String` objects if you prefer, for instance if you need to parse the data yourself.

See samples for such an example.

HL7 Model using HAPI

The HL7 model is Java objects from the HAPI library. Using this library, we can encode and decode from the EDI format (ER7) that is mostly used with HL7. With this model you can code with Java objects instead of the EDI based HL7 format that can be hard for humans to read and understand.

The ER7 sample below is a request to lookup a patient with the patient ID, `0101701234`.

```
MSH|^~\\&|MYSENDER|MYRECEIVER|MYAPPLICATION||200612211200||QRY^A19|1234|P|2.4
QRD|200612211200|R|I|GetPatient|||1^RD|0101701234|DEM||
```

Using the HL7 model we can work with the data as a `ca.uhn.hl7v2.model.Message.Message` object. To retrieve the patient ID for the patient in the ER7 above, you can do this in java code:

```
Message msg = exchange.getIn().getBody(Message.class);
QRD qrd = (QRD)msg.get("QRD");
String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue();
```

Fuse Mediation Router has built-in type converters, so when this operation is invoked:

```
Message msg = exchange.getIn().getBody(Message.class);
```

Fuse Mediation Router will convert the received HL7 data from `String` to `Message`. This is powerful when combined with the HL7 listener, then you as the end-user don't have to work with `byte[]`, `String` or any other simple object formats. You can just use the HAPI HL7 model objects.

Message Headers

The **unmarshal** operation adds these MSH fields as headers on the Camel message:

Camel 1.x

Key	MSH field	Example
`hl7.msh.sendingApplication`	`MSH-3`	`MYSERVER`
`hl7.msh.sendingFacility`	`MSH-4`	`MYSERVERAPP`
`hl7.msh.receivingApplication`	`MSH-5`	`MYCLIENT`
`hl7.msh.receivingFacility`	`MSH-6`	`MYCLIENTAPP`
`hl7.msh.timestamp`	`MSH-7`	`20071231235900`
`hl7.msh.security`	`MSH-8`	`null`
`hl7.msh.messageType`	`MSH-9-1`	`ADT`
`hl7.msh.triggerEvent`	`MSH-9-2`	`A01`
`hl7.msh.messageControl`	`MSH-10`	`1234`
`hl7.msh.processingId`	`MSH-11`	`P`
`hl7.msh.versionId`	`MSH-12`	`2.4`

Camel 2.0

Key	MSH field	Example
`CamelHL7SendingApplication`	`MSH-3`	`MYSERVER`
`CamelHL7SendingFacility`	`MSH-4`	`MYSERVERAPP`
`CamelHL7ReceivingApplication`	`MSH-5`	`MYCLIENT`
`CamelHL7ReceivingFacility`	`MSH-6`	`MYCLIENTAPP`
`CamelHL7Timestamp`	`MSH-7`	`20071231235900`
`CamelHL7Security`	`MSH-8`	`null`
`CamelHL7MessageType`	`MSH-9-1`	`ADT`
`CamelHL7TriggerEvent`	`MSH-9-2`	`A01`

`CamelHL7MessageControl`	`MSH-10`	`1234`
`CamelHL7ProcessingId`	`MSH-11`	`P`
`CamelHL7VersionId`	`MSH-12`	`2.4`

All headers are `String` types. If a header value is missing, its value is `null`.

Options

The HL7 Data Format supports the following options:

Option	Default	Description
validate	true	**Camel 2.0**: Whether the HAPI Parser should validate.

Dependencies

To use HL7 in your camel routes you need to add a dependency on **camel-hl7**, which implements this data format.

If you use Maven, you could just add the following to your `pom.xml`, substituting the version number for the latest & greatest release (see the download page for the latest versions[6]).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hl7</artifactId>
  <version>2.2.0</version>
</dependency>
```

Since HAPI 0.6, the library has been split into a base library[7] and several structures libraries, one for each HL7v2 message version:

- v2.1 structures library[8]
- v2.2 structures library[9]
- v2.3 structures library[10]

[6] Download
[7] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-base/1.0/
[8] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-structures-v21/1.0/
[9] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-structures-v22/1.0/
[10] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-structures-v23/1.0/

- v2.3.1 structures library[11]
- v2.4 structures library[12]
- v2.5 structures library[13]
- v2.5.1 structures library[14]
- v2.6 structures library[15]

By default `camel-hl7` only references the HAPI base library[16]. Applications are responsible for including structures libraries themselves. For example, if a application works with HL7v2 message versions 2.4 and 2.5 then the following dependencies must be added:

```
<dependency>
    <groupId>ca.uhn.hapi</groupId>
    <artifactId>hapi-structures-v24</artifactId>
    <version>1.0</version>
</dependency>
<dependency>
    <groupId>ca.uhn.hapi</groupId>
    <artifactId>hapi-structures-v25</artifactId>
    <version>1.0</version>
</dependency>
```

OSGi

An OSGi bundle containing the base library, all structures libraries and required dependencies (on the bundle classpath) can be downloaded from the HAPI Maven repository[17] as well.

```
<dependency>
    <groupId>ca.uhn.hapi</groupId>
    <artifactId>hapi-osgi-base</artifactId>
    <version>1.0.1</version>
</dependency>
```

[11] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-structures-v231/1.0/
[12] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-structures-v24/1.0/
[13] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-structures-v24/1.0/
[14] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-structures-v251/1.0/
[15] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-structures-v26/1.0/
[16] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-base/1.0/
[17] http://hl7api.sourceforge.net/m2/ca/uhn/hapi/hapi-osgi-base/

Samples

In the following example we send a HL7 request to a HL7 listener and retrieves a response. We use plain `String` types in this example:

```
String line1 = "MSH|^~\\&|MYSENDER|MYRECEIVER|MYAPPLICA
TION||200612211200||QRY^A19|1234|P|2.4";
String line2 = "QRD|200612211200|R|I|GetPatient|||1^RD|0101701234|DEM||";

StringBuilder in = new StringBuilder();
in.append(line1);
in.append("\n");
in.append(line2);

String out = (String)template.requestBody("mina:tcp://127.0.0.1:8888?sync=true&codec=#hl7co
dec", in.toString());
```

In the next sample, we want to route HL7 requests from our HL7 listener to our business logic. We have our business logic in a plain POJO that we have registered in the registry as `hl7service` = for instance using Spring and letting the bean id = `hl7service`.

Our business logic is a plain POJO only using the HAPI library so we have these operations defined:

```
public class MyHL7BusinessLogic {

    // This is a plain POJO that has NO imports whatsoever on Fuse Mediation Router.
    // its a plain POJO only importing the HAPI library so we can much easier work with the
 HL7 format.

    public Message handleA19(Message msg) throws Exception {
        // here you can have your business logic for A19 messages
        assertTrue(msg instanceof QRY_A19);
        // just return the same dummy response
        return createADR19Message();
    }

    public Message handleA01(Message msg) throws Exception {
        // here you can have your business logic for A01 messages
        assertTrue(msg instanceof ADT_A01);
        // just return the same dummy response
        return createADT01Message();
    }
}
```

Then we set up the Fuse Mediation Router routes using the `RouteBuilder` as follows:

```
DataFormat hl7 = new HL7DataFormat();
// we setup or HL7 listener on port 8888 (using the hl7codec) and in sync mode so we can
```

```
return a response
from("mina:tcp://127.0.0.1:8888?sync=true&codec=#hl7codec")
    // we use the HL7 data format to unmarshal from HL7 stream to the HAPI Message model
    // this ensures that the camel message has been enriched with hl7 specific headers to
    // make the routing much easier (see below)
    .unmarshal(hl7)
    // using choice as the content base router
    .choice()
        // where we choose that A19 queries invoke the handleA19 method on our hl7service
bean
        .when(header("CamelHL7TriggerEvent").isEqualTo("A19"))
            .beanRef("hl7service", "handleA19")
            .to("mock:a19")
        // and A01 should invoke the handleA01 method on our hl7service bean
        .when(header("CamelHL7TriggerEvent").isEqualTo("A01")).to("mock:a01")
            .beanRef("hl7service", "handleA01")
            .to("mock:a19")
        // other types should go to mock:unknown
        .otherwise()
            .to("mock:unknown")
    // end choice block
    .end()
    // marhsal response back
    .marshal(hl7);
```

Notice that we use the HL7 DataFormat to enrich our Camel Message with the MSH fields preconfigued on the Camel Message. This lets us much more easily define our routes using the fluent builders. If we do not use the HL7 DataFormat, then we do not gains these headers and we must resort to a different technique for computing the MSH trigger event (= what kind of HL7 message it is). This is a big advantage of the HL7 DataFormat over the plain HL7 type converters.

Sample using plain String objects

In this sample we use plain `String` objects as the data format, that we send, process and receive. As the sample is part of a unit test, there is some code for assertions, but you should be able to understand what happens. First we send the plain string, `Hello World`, to the `HL7MLLPCodec` and receive the response as a plain string, `Bye World`.

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedBodiesReceived("Bye World");

// send plain hello world as String
Object out = template.requestBody("mina:tcp://127.0.0.1:8888?sync=true&codec=#hl7codec",
"Hello World");

assertMockEndpointsSatisfied();
```

```
// and the response is also just plain String
assertEquals("Bye World", out);
```

Here we process the incoming data as plain String and send the response also as plain String:

```
from("mina:tcp://127.0.0.1:8888?sync=true&codec=#hl7codec")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            // use plain String as message format
            String body = exchange.getIn().getBody(String.class);
            assertEquals("Hello World", body);

            // return the response as plain string
            exchange.getOut().setBody("Bye World");
        }
    })
    .to("mock:result");
```

Chapter 34. HTTP

HTTP Component

The **http:** component provides HTTP based endpoints[1] for consuming external HTTP resources (as a client to call external servers using HTTP).

URI format

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

Will by default use port 80 for HTTP and 443 for HTTPS.

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

camel-http vs camel-jetty

You can only produce to endpoints generated by the HTTP component. Therefore it should never be used as input into your Fuse Mediation Router Routes. To bind/expose an HTTP endpoint via a HTTP server as input to a camel route, you can use the Jetty Component on page 277

HttpEndpoint Options

Name	Default Value	Description
`throwExceptionOnFailure`	`true`	**Fuse Mediation Router 2.0:** Option to disable throwing the `HttpOperationFailedException` in case of failed responses from the remote server. This allows you to get all responses regardles of the HTTP status code.
`bridgeEndpoint`	`false`	**Camel 2.1:** If the option is true , HttpProducer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for request. You may also set the **throwExcpetionOnFailure** to be false to let the HttpProducer send all the fault response back. **Camel 2.3:** If the option is true, HttpProducer and CamelServlet will skip the gzip processing if the content-encoding is "gzip".

[1] Endpoint

`disableStreamCache`	`false`	**Camel 2.3:** DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support read it twice, otherwise DefaultHttpBinding will set the request input stream direct into the message body.
`httpBindingRef`	`null`	Reference to a `org.apache.camel.component.http.HttpBinding` in the Registry.
`httpBinding`	`null`	**Camel 2.3:** Reference to a `org.apache.camel.component.http.HttpBinding` in the Registry.
`httpClientConfigurerRef`	`null`	Reference to a `org.apache.camel.component.http.HttpClientConfigurer` in the Registry. From Camel 2.3 onwards prefer to use the `httpClientConfigurer` option.
`httpClientConfigurer`	`null`	**Camel 2.3:** Reference to a `org.apache.camel.component.http.HttpClientConfigurer` in the Registry.
`httpClient.XXX`	`null`	Setting options on the HttpClientParams[2]. For instance `httpClient.soTimeout=5000` will set the `SO_TIMEOUT` to 5 seconds.
`clientConnectionManager`	`null`	**Camel 2.3:** To use a custom `org.apache.http.conn.ClientConnectionManager`.
`transferException`	`false`	**Camel 2.6:** If enabled and an Exchange failed processing on the consumer side, and if the caused `Exception` was send back serialized in the response as a `application/x-java-serialized-object` content type (for example using Jetty on page 277 or Servlet Camel components). On the producer side the exception will be deserialized and thrown as is, instead of the `HttpOperationFailedException`. The caused exception is required to be serialized.

The following authentication options can also be set on the HttpEndpoint:

Name	Default Value	Description
`authMethod`	`null`	Authentication method, either as `Basic`, `Digest` or `NTLM`.
`authMethodPriority`	`null`	Priority of authentication methods. Is a list separated with comma. For example: `Basic,Digest` to exclude `NTLM`.
`authUsername`	`null`	Username for authentication
`authPassword`	`null`	Password for authentication
`authDomain`	`null`	Domain for NTML authentication

[2] http://hc.apache.org/httpclient-3.x/apidocs/org/apache/commons/httpclient/params/HttpClientParams.html

`authHost`	`null`	Optional host for NTML authentication
`proxyHost`	`null`	The proxy host name
`proxyPort`	`null`	The proxy port number
`proxyAuthMethod`	`null`	Authentication method for proxy, either as `Basic`, `Digest` or `NTLM`.
`proxyAuthUsername`	`null`	Username for proxy authentication
`proxyAuthPassword`	`null`	Password for proxy authentication
`proxyAuthDomain`	`null`	Domain for proxy NTML authentication
`proxyAuthHost`	`null`	Optional host for proxy NTML authentication

When using authentication you **must** provide the choice of method for the `authMethod` or `authProxyMethod` options. You can configure the proxy and authentication details on either the `HttpComponent` or the `HttpEndoint`. Values provided on the `HttpEndpoint` will take precedence over `HttpComponent`. Its most likely best to configure this on the `HttpComponent` which allows you to do this once.

The Http component uses convention over configuration which means that if you have not explicit set a `authMethodPriority` then it will fallback and use the select(ed) `authMethod` as priority as well. So if you use `authMethod.Basic` then the `auhtMethodPriority` will be `Basic` only.

Exchange Properties Fuse Mediation Router 2.x

The following `Exchange` properties are recognized by HTTP endpoints:

Name	Type	Description
`Exchange.HTTP_URI`	`String`	URI to call. Will override existing URI set directly on the endpoint.
`Exchange.HTTP_PATH`	`String`	Request URI's path, the header will be used to build the request URI with the HTTP_URI. **Camel 2.3.0:** If the path is start with "/", http producer will try to find the relative path based on the Exchange.HTTP_BASE_URI header or the exchange.getFromEndpoint().getEndpointUri();
`Exchange.HTTP_QUERY`	`String`	URI parameters. Will override existing URI parameters set directly on the endpoint.
`Exchange.HTTP_RESPONSE_CODE`	`int`	The HTTP response code from the external server. Is 200 for OK.
`Exchange.HTTP_CHARACTER_ENCODING`	`String`	Character encoding.

`Exchange.CONTENT_TYPE`	`String`	The HTTP content type. Is set on both the IN and OUT message to provide a content type, such as `text/html`.
`Exchange.CONTENT_ENCODING`	`String`	The HTTP content encoding. Is set on both the IN and OUT message to provide a content encoding, such as `gzip`.
`Exchange.HTTP_SERVLET_REQUEST`	`HttpServletRequest`	**Fuse Mediation Router 2.3:** The `HttpServletRequest` object.
`Exchange.HTTP_SERVLET_RESPONSE`	`HttpServletResponse`	**Fuse Mediation Router 2.3:** The `HttpServletResponse` object.
`Exchange.HTTP_PROTOCOL_VERSION`	`String`	**Camel 2.5:** You can set the http protocol version with this header, eg. "HTTP/1.0". If you didn't specify the header, HttpProducer will use the default value "HTTP/1.1"

Message Body

Fuse Mediation Router will store the HTTP response from the external server on the OUT body. All headers from the IN message will be copied to the OUT message, so headers are preserved during routing. Additionally Fuse Mediation Router will add the HTTP response headers as well to the OUT message headers.

Response code

Fuse Mediation Router will handle according to the HTTP response code:

- Response code is in the range 100..299, Fuse Mediation Router regards it as a success response.
- Response code is in the range 300..399, Fuse Mediation Router regards it as a redirection response and will throw a `HttpOperationFailedException` with the information.
- Response code is 400+, Fuse Mediation Router regards it as an external server failure and will throw a `HttpOperationFailedException` with the information.The option, `throwExceptionOnFailure`, can be set to `false` to prevent the `HttpOperationFailedException` from being thrown for failed response codes. This allows you to get any response from the remote server.There is a sample below demonstrating this.

HttpOperationFailedException

This exception contains the following information:

- The HTTP status code

- The HTTP status line (text of the status code)
- Redirect location, if server returned a redirect
- Response body as a `java.lang.String`, if server provided a body as response

Calling using GET or POST

From **Fuse Mediation Router 1.5** onwards, the following algorithm is used to determine if either `GET` or `POST` HTTP method should be used:

1. Use method provided in header.
2. `GET` if query string is provided in header.
3. `GET` if endpoint is configured with a query string.
4. `POST` if there is data to send (body is not null).
5. `GET` otherwise.

How to get access to HttpServletRequest and HttpServletResponse

Available as of Fuse Mediation Router 2.0

You can get access to these two using the Camel type converter system using **NOTE** from Camel 2.3.0 you can get the request and response not just from the processor after the camel-jetty or camel-cxf endpoint.

```
HttpServletRequest request = exchange.getIn().getBody(HttpServletRequest.class);
HttpServletRequest response = exchange.getIn().getBody(HttpServletResponse.class);
```

Configuring URI to call

You can set the HTTP producer's URI directly form the endpoint URI. In the route below, Fuse Mediation Router will call out to the external server, `oldhost`, using HTTP.

```
from("direct:start")
     .to("http://oldhost");
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="http://oldhost"/>
  </route>
</camelContext>
```

In **Fuse Mediation Router 1.5.1** you can override the HTTP endpoint URI by adding a header with the key, `HttpProducer.HTTP_URI`, on the message.

```
from("direct:start")
            .setHeader(org.apache.camel.component.http.HttpProducer.HTTP_URI, constant("ht
tp://newhost"))
     .to("http://oldhost");
```

In the sample above Fuse Mediation Router will call the http://newhost [3] despite the endpoint is configured with http://oldhost [4].

In **Fuse Mediation Router 2.0**, you can override the HTTP endpoint URI by setting the `Exchange.HTTP_URI` header, as follows:

```
from("direct:start")
            .setHeader(Exchange.HTTP_URI, constant("http://newhost"))
     .to("http://oldhost");
```

Configuring URI Parameters

The **http** producer supports URI parameters to be sent to the HTTP server. The URI parameters can either be set directly on the endpoint URI, as follows:

```
from("direct:start")
     .to("http://oldhost?order=123&detail=short");
```

Or as a header with the key, `Exchange.HTTP_QUERY`, on the message, as follows:

```
from("direct:start")
            .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
     .to("http://oldhost");
```

[3] http://newhost
[4] http://oldhost

How to set the http method (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE) to the HTTP producer

The HTTP component provides a way to set the HTTP request method by setting the message header. Here is an example;

```
from("direct:start")
            .setHeader(Exchange.HTTP_METHOD, constant(org.apache.camel.component.http.Http
Methods.POST))
     .to("http://www.google.com")
            .to("mock:results");
```

The method can be written a bit shorter using the string constants:

```
.setHeader("CamelHttpMethod", constant("POST"))
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <setHeader headerName="CamelHttpMethod">
        <constant>POST</constant>
    </setHeader>
    <to uri="http://www.google.com"/>
    <to uri="mock:results"/>
  </route>
</camelContext>
```

Using client tineout - SO_TIMEOUT

See the unit test in this link[5]

Configuring a Proxy

Only for >= Fuse Mediation Router 1.6.2 The HTTP component provides a way to configure a proxy.

```
from("direct:start")
     .to("http://oldhost?proxyHost=www.myproxy.com&proxyPort=80");
```

There is also support for proxy authentication via the `proxyUsername` and `proxyPassword` options.

[5] http://svn.apache.org/viewvc?view=rev&revision=781775

Using proxy settings outside of URI

Only for >= Fuse Mediation Router 1.6.2 The HTTP component will detect Java System Properties for `http.proxyHost` and `http.proxyPort` and use them if provided. See more at SUN http proxy documentation[6].

To avoid the System properties conflicts, from Fuse Mediation Router 2.2.0 you can only set the proxy configure from CameContext or URI. Java DSL :

```
context.getProperties().put("http.proxyHost", "172.168.18.9");
context.getProperties().put("http.proxyPort" "8080");
```

Spring XML

```
  <camelContext>
      <properties>
          <property key="http.proxyHost" value="172.168.18.9"/>
          <property key="http.proxyPort" value="8080"/>
     </properties>
  </camelContext>
```

Fuse Mediation Router will first set the settings from Java System or CamelContext Properties and then the endpoint proxy options if provided. So you can override the system properties with the endpoint options.

Configuring charset

If you are using `POST` to send data you can configure the `charset` using the `Exchange` property:

```
exchange.setProperty(Exchange.CHARSET_NAME, "iso-8859-1");
```

Or the httpClient options: `httpClient.contentCharset=iso-8859-1`

Sample with scheduled poll

The sample polls the Google homepage every 10 seconds and write the page to the file `message.html`:

```
from("timer://foo?fixedRate=true&delay=0&period=10000")
    .to("http://www.google.com")
    .setHeader(FileComponent.HEADER_FILE_NAME, "message.html").to("file:target/google");
```

[6] http://java.sun.com/javase/6/docs/technotes/guides/net/proxies.html

URI Parameters from the endpoint URI

In this sample we have the complete URI endpoint that is just what you would have typed in a Web browser. Multiple URI parameters can of course be set using the `&` character as separator, just as you would in the web browser. Fuse Mediation Router does no tricks here.

```
// we query for Camel at the Google page
template.sendBody("http://www.google.com/search?q=Camel", null);
```

URI Parameters from the Message

```
Map headers = new HashMap();
headers.put(Exchange.HTTP_QUERY, "q=Camel&lr=lang_en");
// we query for Camel and English language at Google
template.sendBody("http://www.google.com/search", null, headers);
```

In the header value above notice that it should **not** be prefixed with `?` and you can separate parameters as usual with the `&` char.

Getting the Response Code

You can get the HTTP response code from the HTTP component by getting the value from the Out message header with `Exchange.HTTP_RESPONSE_CODE`.

```
Exchange exchange = template.send("http://www.google.com/search", new Processor() {
            public void process(Exchange exchange) throws Exception {
              exchange.getIn().setHeader(Exchange.HTTP_QUERY, constant("hl=en&q=activemq"));

            }
   });
   Message out = exchange.getOut();
   int responseCode = out.getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);
```

Using throwExceptionOnFailure=false to get any response back

Available as of Fuse Mediation Router 2.0 In the route below we want to route a message that we enrich with data returned from a remote HTTP call. As we want any response from the remote server, we set the `throwExceptionOnFailure` option to `false` so we get any response in the `AggregationStrategy`. As the code is based on a unit test that simulates a HTTP status code 404, there is some assertion code etc.

```
// We set throwExceptionOnFailure to false to let Fuse Mediation Router return any response
 from the remove HTTP server without thrown
// HttpOperationFailedException in case of failures.
// This allows us to handle all responses in the aggregation strategy where we can check
```

```
the HTTP response code
// and decide what to do. As this is based on an unit test we assert the code is 404
from("direct:start").enrich("http://localhost:8222/myserver?throwExceptionOnFail
ure=false&user=Camel", new AggregationStrategy() {
    public Exchange aggregate(Exchange original, Exchange resource) {
        // get the response code
       Integer code = resource.getIn().getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);

        assertEquals(404, code.intValue());
        return resource;
    }
}).to("mock:result");

// this is our jetty server where we simulate the 404
from("jetty://http://localhost:8222/myserver")
        .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                exchange.getOut().setBody("Page not found");
                exchange.getOut().setHeader(Exchange.HTTP_RESPONSE_CODE, 404);
            }
        });
```

Disabling Cookies

To disable cookies you can set the HTTP Client to ignore cookies by adding this URI option: `httpClient.cookiePolicy=ignoreCookies`

Advanced Usage

If you need more control over the HTTP producer you should use the `HttpComponent` where you can set various classes to give you custom behavior.

Setting MaxConnectionsPerHost

The Http Component has a `org.apache.commons.httpclient.HttpConnectionManager` where you can configure various global configuration for the given component. By global, we mean that any endpoint the component creates has the same shared `HttpConnectionManager`. So, if we want to set a different value for the max connection per host, we need to define it on the HTTP component and **not** on the endpoint URI that we usually use. So here comes:

First, we define the `http` component in Spring XML. Yes, we use the same scheme name, `http`, because otherwise Fuse Mediation Router will auto-discover and create the component with default settings. What we need is to overrule this so we can set our options. In the sample below we set the max connection to 5 instead of the default of 2.

```
<bean id="http" class="org.apache.camel.component.http.HttpComponent">
    <property name="camelContext" ref="camel"/>
    <property name="httpConnectionManager" ref="myHttpConnectionManager"/>
</bean>

<bean id="myHttpConnectionManager" class="org.apache.commons.httpclient.MultiThreadedHttp
ConnectionManager">
    <property name="params" ref="myHttpConnectionManagerParams"/>
</bean>

<bean id="myHttpConnectionManagerParams" class="org.apache.commons.httpclient.params.Http
ConnectionManagerParams">
    <property name="defaultMaxConnectionsPerHost" value="5"/>
</bean>
```

And then we can just use it as we normally do in our routes:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring" trace="true">
    <route>
        <from uri="direct:start"/>
        <to uri="http://www.google.com"/>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

Using HTTPS to authenticate

Some HTTPS servers do not return a HTTP code 401 Authorization Required, which can cause HTTPS connections to fail. The solution to this problem is to set the following URI option: `httpClient.authenticationPreemptive=true`.

Setting up SSL for HTTP Client

Basically `camel-http` component is built on the top of Apache HTTP client, and you can implement a custom `org.apache.camel.component.http.HttpClientConfigurer` to do some configuration on the HTTP client if you need full control of it.

However if you *just* want to specify the keystore and truststore you can do this with Apache HTTP `HttpClientConfigurer`, for example:

```
Protocol authhttps = new Protocol("https", new AuthSSLProtocolSocketFactory(
  new URL("file:my.keystore"), "mypassword",
  new URL("file:my.truststore"), "mypassword"), 443);

Protocol.registerProtocol("https", authhttps);
```

And then you need to create a class that implements `HttpClientConfigurer`, and registers https protocol providing a keystore or truststore per example above. Then, from your Fuse Mediation Router route builder class you can hook it up like so:

```
HttpComponent httpComponent = getContext().getComponent("http", HttpComponent.class);
httpComponent.setHttpClientConfigurer(new MyHttpClientConfigurer());
```

If you are doing this using the Spring DSL, you can specify your `HttpClientConfigurer` using the URI. For example:

```
<bean id="myHttpClientConfigurer"
 class="my.https.HttpClientConfigurer">
</bean>

<to uri="https://myhostname.com:443/myURL?httpClientConfigurerRef=myHttpClientConfigurer"/>
```

As long as you implement the HttpClientConfigurer and configure your keystore and truststore as described above, it will work fine.

See also:

- Jetty on page 277

Chapter 35. iBATIS

iBATIS

The **ibatis:** component allows you to query, poll, insert, update and delete data in a relational database using Apache iBATIS[1].

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

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

URI format

```
ibatis:statementName[?options]
```

Where **statementName** is the name in the iBATIS XML configuration file which maps to the query, insert, update or delete operation you wish to evaluate.

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

This component will by default load the iBatis SqlMapConfig file from the root of the classpath and expected named as `SqlMapConfig.xml`. It uses Spring resource loading so you can define it using `classpath`, `file` or `http` as prefix to load resources with those schemes. In Camel 2.2 you can configure this on the iBatisComponent with the `setSqlMapConfig(String)` method.

Options

Option	Type	Default	Description
`consumer.onConsume`	`String`	`null`	Statements to run after consuming. Can be used, for example, to update rows after they have been consumed and processed in Fuse Mediation Router. See sample later. Multiple statements can be separated with comma.

[1] http://ibatis.apache.org/

`consumer.useIterator`	`boolean`	`true`	If `true` each row returned when polling will be processed individually. If `false` the entire `List` of data is set as the IN body.
`consumer.routeEmptyResultSet`	`boolean`	`false`	**Fuse Mediation Router 2.0:** Sets whether empty result set should be routed or not. By default, empty result sets are not routed.
`statementType`	`StatementType`	`null`	**Fuse Mediation Router 1.6.1/2.0:** Mandatory to specify for IbatisProducer to control which iBatis `SqlMapClient` method to invoke. The enum values are: `QueryForObject`, `QueryForList`, `Insert`, `Update`, `Delete`.
`maxMessagesPerPoll`	`int`	`0`	**Fuse Mediation Router 2.0:** An integer to define a maximum messages to gather per poll. By default, no maximum is set. Can be used to set a limit of e.g. 1000 to avoid when starting up the server that there are thousands of files. Set a value of 0 or negative to disabled it.

Message Headers

Fuse Mediation Router will populate the result message, either IN or OUT with a header with the operationName used:

Header	Type	Description
`org.apache.camel.ibatis.queryName`	`String`	Fuse Mediation Router 1.x: The **statementName** used (for example: insertAccount).
`CamelIBatisStatementName`	`String`	Fuse Mediation Router 2.0: The **statementName** used (for example: insertAccount).
`CamelIBatisResult`	`Object`	Fuse Mediation Router 1.6.2/2.0: The **response** returned from iBatis in any of the operations. For instance an `INSERT` could return the auto-generated key, or number of rows etc.

Message Body

Fuse Mediation Router 1.6.1: The response from iBatis will be set as OUT body.

Fuse Mediation Router 1.6.2/2.0: The response from iBatis will only be set as body if it's a `SELECT` statement. That means, for example, for `INSERT` statements Fuse Mediation Router will not replace the body. This allows

you to continue routing and keep the original body. The response from iBatis is always stored in the header with the key `CamelIBatisResult`.

Samples

For example if you wish to consume beans from a JMS queue and insert them into a database you could do the following:

```
from("activemq:queue:newAccount").
  to("ibatis:insertAccount?statementType=Insert");
```

Notice we have to specify the `statementType`, as we need to instruct Fuse Mediation Router which `SqlMapClient` operation to invoke.

Where **insertAccount** is the iBatis ID in the SQL map file:

```
  <!-- Insert example, using the Account parameter class -->
  <insert id="insertAccount" parameterClass="Account">
    insert into ACCOUNT (
      ACC_ID,
      ACC_FIRST_NAME,
      ACC_LAST_NAME,
      ACC_EMAIL
    )
    values (
      #id#, #firstName#, #lastName#, #emailAddress#
    )
  </insert>
```

Using StatementType for better control of IBatis

Available as of Fuse Mediation Router 1.6.1/2.0 When routing to an iBatis endpoint you want more fine grained control so you can control whether the SQL statement to be executed is a `SELEECT`, `UPDATE`, `DELETE` or `INSERT` etc. This is now possible in Fuse Mediation Router 1.6.1/2.0. So for instance if we want to route to an iBatis endpoint in which the IN body contains parameters to a `SELECT` statement we can do:

```
from("direct:start")
    .to("ibatis:selectAccountById?statementType=QueryForObject")
    .to("mock:result");
```

In the code above we can invoke the iBatis statement `selectAccountById` and the IN body should contain the account id we want to retrieve, such as an `Integer` type.

We can do the same for some of the other operations, such as `QueryForList`:

```
from("direct:start")
    .to("ibatis:selectAllAccounts?statementType=QueryForList")
    .to("mock:result");
```

And the same for `UPDATE`, where we can send an `Account` object as IN body to iBatis:

```
from("direct:start")
    .to("ibatis:updateAccount?statementType=Update")
    .to("mock:result");
```

Scheduled polling example

Since this component does not support scheduled polling, you need to use another mechanism for triggering the scheduled polls, such as the Timer on page 507 or Quartz on page 389 components.

In the sample below we poll the database, every 30 seconds using the Timer on page 507 component and send the data to the JMS queue:

```
from("timer://pollTheDatabase?delay=30000").to("ibatis:selectAllAccounts?statementType=Query
ForList").to("activemq:queue:allAccounts");
```

And the iBatis SQL map file used:

```
  <!-- Select with no parameters using the result map for Account class. -->
  <select id="selectAllAccounts" resultMap="AccountResult">
    select * from ACCOUNT
  </select>
```

Using onConsume

This component supports executing statements **after** data have been consumed and processed by Fuse Mediation Router. This allows you to do post updates in the database. Notice all statements must be `UPDATE` statements. Fuse Mediation Router supports executing multiple statements whose name should be separated by comma.

The route below illustrates we execute the **consumeAccount** statement data is processed. This allows us to change the status of the row in the database to processed, so we avoid consuming it twice or more.

```
from("ibatis:selectUnprocessedAccounts?consumer.onConsume=consumeAccount").to("mock:results");
```

And the statements in the sqlmap file:

```
<select id="selectUnprocessedAccounts" resultMap="AccountResult">
     select * from ACCOUNT where PROCESSED = false
</select>
```

```
<update id="consumeAccount" parameterClass="Account">
     update ACCOUNT set PROCESSED = true where ACC_ID = #id#
</update>
```

Chapter 36. IRC

IRC Component

The **irc** component implements an IRC[1] (Internet Relay Chat) transport.

URI format

```
irc:nick@host[:port]/#room[?options]
```

In Fuse Mediation Router 2.0, you can also use the following format:

```
irc:nick@host[:port]?channels=#channel1,#channel2,#channel3[?options]
```

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

Options

Name	Description	Example	Default Value
`channels`	New in 2.0, comma separated list of IRC channels to join.	`channels=#channel1,#channel2`	`null`
`nickname`	The nickname used in chat.	`irc:MyNick@irc.server.org#channel` or `irc:irc.server.org#channel?nickname=MyUser`	`null`
`username`	The IRC server user name.	`irc:MyUser@irc.server.org#channel` or `irc:irc.server.org#channel?username=MyUser`	Same as nickname.
`password`	The IRC server password.	`password=somepass`	*None*
`realname`	The IRC user's actual name.	`realname=MyName`	*None*
`colors`	Whether or not the server supports color codes.	`true, false`	`true`
`onReply`	Whether or not to handle general responses to commands or	`true, false`	`false`

[1] http://en.wikipedia.org/wiki/Internet_Relay_Chat

	informational messages.		
`onNick`	Handle nickname change events.	`true, false`	`true`
`onQuit`	Handle user quit events.	`true, false`	`true`
`onJoin`	Handle user join events.	`true, false`	`true`
`onKick`	Handle kick events.	`true, false`	`true`
`onMode`	Handle mode change events.	`true, false`	`true`
`onPart`	Handle user part events.	`true, false`	`true`
`onTopic`	Handle topic change events.	`true, false`	`true`
`onPrivmsg`	Handle message events.	`true, false`	`true`
`trustManager`	New in 2.0, the trust manager used to verify the SSL server's certificate.	`trustManager=#referenceToTrustManagerBean`	The default trust manager, which accepts *all* certificates, will be used.
`keys`	**Camel 2.2:** Comma separated list of IRC channel keys. Important to be listed in same order as channels. When joining multiple channels with only some needing keys just insert an empty value for that channel.	`irc:MyNick@irc.server.org/#channel?keys=chankey`	`null`

SSL Support

As of Fuse Mediation Router 2.0, you can also connect to an SSL enabled IRC server, as follows:

```
ircs:host[:port]/#room?username=user&password=pass
```

By default, the IRC transport uses SSLDefaultTrustManager[2]. If you need to provide your own custom trust manager, use the `trustManager` parameter as follows:

```
ircs:host[:port]/#room?username=user&password=pass&trustManager=#referenceToMyTrustManagerBean
```

Using keys

Available as of Camel 2.2 Some irc rooms requires you to provide a key to be able to join that channel. The key is just a secret word.

For example we join 3 channels where as only channel 1 and 3 uses a key.

```
irc:nick@irc.server.org?channels=#chan1,#chan2,#chan3&keys=chan1Key,,chan3key
```

[2] http://moepii.sourceforge.net/irclib/javadoc/org/schwering/irc/lib/ssl/SSLDefaultTrustManager.html

Chapter 37. JavaSpace

JavaSpace Component

The **javaspace** component is a transport for working with any JavaSpace compliant implementation and this component has been tested with both the Blitz implementation[1] and the GigaSpace implementation[2]. This component can be used for sending and receiving any object inheriting from the Jini `net.jini.core.entry.Entry` class. It is also possible to pass the bean ID of a template that can be used for reading/taking the entries from the space. This component can be used for sending/receiving any serializable object acting as a sort of generic transport. The JavaSpace component contains a special optimization for dealing with the `BeanExchange`. It can be used to invoke a POJO remotely, using a JavaSpace as a transport. This latter feature can provide a simple implementation of the master/worker pattern, where a POJO provides the business logic for the worker. Look at the test cases for examples of various use cases for this component.

URI format

```
javaspace:jini://host[?options]
```

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

Options

Name	Default Value	Description
`spaceName`	`null`	Specifies the JavaSpace name.
`verb`	`take`	Specifies the verb for getting JavaSpace entries. The values can be: `take` or `read`.
`transactional`	`false`	If `true`, sending and receiving entries is performed within a transaction.
`transactionalTimeout`	`Long.MAX_VALUE`	Specifies the transaction timeout.
`concurrentConsumers`	`1`	Specifies the number of concurrent consumers getting entries from the JavaSpace.
`templateId`	`null`	If present, this option specifies the Spring bean ID of the template to use for reading/taking entries.

[1] http://www.dancres.org/blitz/
[2] http://www.gigaspaces.com/

Sending and Receiving Entries

```
//Sending route
from("direct:input").to("javaspace:jini://localhost?spaceName=mySpace");

//Receiving Route
from("javaspace:jini://localhost?spaceName=mySpace&templateId=template&verb=take&concurrent
Consumers=1")
```

In this case the payload can be any object that inherits from the Jini `Entry` type.

Sending and receiving serializable objects

Using the preceding routes, it is also possible to send and receive any serializable object. The JavaSpace component detects that the payload is not a Jini `Entry` and then it automatically wraps the payload with a Camel Jini `Entry`. In this way, a JavaSpace can be used as a generic transport mechanism.

Using JavaSpace as a remote invocation transport

The JavaSpace component has been tailored to work in combination with the Camel bean component. It is therefore possible to call a remote POJO using JavaSpace as the transport:

```
from("direct:input").to("javaspace:jini://localhost?spaceName=mySpace"); //Client side

from("javaspace:jini://localhost?concurrentConsumers=10&spaceName=mySpace").to("pojo:pojo");
 //Server side
```

In the code there are two test cases showing how to use a POJO to realize the master/worker pattern. The idea is to use the POJO to provide the business logic and rely on Fuse Mediation Router for sending/receiving requests/replies with the proper correlation.

Chapter 38. Jasypt

Jasypt component

Available as of Camel 2.5

Jasypt[1] is a simplified encryption library which makes encryption and decryption easy. Camel integrates with Jasypt to allow sensitive information in Properties on page 387 files to be encrypted. By dropping **`camel-jasypt`** on the classpath those encrypted values will automatic be decrypted on-the-fly by Camel. This ensures that human eyes can't easily spot sensitive information such as usernames and passwords.

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

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

Tooling

The Jasypt on page 253 component provides a little command line tooling to encrypt or decrypt values.

The console output the syntax and which options it provides:

```
Apache Camel Jasypt takes the following options

  -h or -help = Displays the help screen
  -c or -command <command> = Command either encrypt or decrypt
  -p or -password <password> = Password to use
  -i or -input <input> = Text to encrypt or decrypt
  -a or -algorithm <algorithm> = Optional algorithm to use
```

For example to encrypt the value `tiger` you run with the following parameters. In the apache camel kit, you cd into the lib folder and run the following java cmd, where <*CAMEL*HOME>_ is where you have downloaded and extract the Camel distribution.

[1] http://www.jasypt.org/

```
$ cd <CAMEL_HOME>/lib
$ java -jar camel-jasypt-2.5.0.jar -c encrypt -p secret -i tiger
```

Which outputs the following result

```
Encrypted text: qaEEacuW7BUti8LcMgyjKw==
```

This means the encrypted representation `qaEEacuW7BUti8LcMgyjKw==` can be decrypted back to `tiger` if you know the master password which was `secret`. If you run the tool again then the encrypted value will return a different result. But decrypting the value will always return the correct original value.

So you can test it by running the tooling using the following parameters:

```
$ cd <CAMEL_HOME>/lib
$ java -jar camel-jasypt-2.5.0.jar -c decrypt -p secret -i qaEEacuW7BUti8LcMgyjKw==
```

Which outputs the following result:

```
Decrypted text: tiger
```

The idea is then to use those encrypted values in your Properties on page 387 files. Notice how the password value is encrypted and the value has the tokens surrounding `ENC(value here)`

```
# refer to a mock endpoint name by that encrypted password
cool.result=mock:{{cool.password}}

# here is a password which is encrypted
cool.password=ENC(bsW9uV37gQ0QHFu7KO03Ww==)
```

Tooling dependencies

The tooling requires the following JARs in the classpath, which has been enlisted in the `MANIFEST.MF` file of `camel-jasypt` with `optional/` as prefix. Hence why the java cmd above can pickup the needed JARs from the Apache Distribution in the `optional` directory.

```
jasypt-1.6.jar commons-lang-2.4.jar commons-codec-1.4.jar icu4j-4.0.1.jar
```

Java 1.5 users

The `icu4j-4.0.1.jar` is only needed when running on JDK 1.5.

This JAR is not distributed by Apache Camel and you have to download it manually and copy it to the `lib/optional` directory of the Camel distribution. You can download it from Apache Central Maven repo[2].

URI Options

The options below are exclusive for the Jasypt on page 253 component.

Name	Default Value	Type	Description
`password`	`null`	`String`	Specifies the master password to use for decrypting. This option is mandatory. See below for more details.
`algorithm`	`null`	`String`	Name of an optional algorithm to use.

Protecting the master password

The master password used by Jasypt on page 253 must be provided, so its capable of decrypting the values. However having this master password out in the opening may not be an ideal solution. Therefore you could for example provided it as a JVM system property or as a OS environment setting. If you decide to do so then the `password` option supports prefixes which dictates this. `sysenv:` means to lookup the OS system environment with the given key. `sys:` means to lookup a JVM system property.

For example you could provided the password before you start the application

```
$ export CAMEL_ENCRYPTION_PASSWORD=secret
```

Then start the application, such as running the start script.

When the application is up and running you can unset the environment

```
$ unset CAMEL_ENCRYPTION_PASSWORD
```

The `password` option is then a matter of defining as follows: `password=sysenv:CAMEL_ENCRYPTION_PASSWORD`.

[2] http://repo2.maven.org/maven2/com/ibm/icu/icu4j/4.0.1/

Example with Java DSL

In Java DSL you need to configure Jasypt on page 253 as a `JasyptPropertiesParser` instance and set it on the Properties on page 387 component as show below:

```
// create the jasypt properties parser
JasyptPropertiesParser jasypt = new JasyptPropertiesParser();
// and set the master password
jasypt.setPassword("secret");

// create the properties component
PropertiesComponent pc = new PropertiesComponent();
pc.setLocation("classpath:org/apache/camel/component/jasypt/myproperties.properties");
// and use the jasypt properties parser so we can decrypt values
pc.setPropertiesParser(jasypt);

// add properties component to camel context
context.addComponent("properties", pc);
```

The properties file `myproperties.properties` then contain the encrypted value, such as shown below. Notice how the password value is encrypted and the value has the tokens surrounding `ENC(value here)`

```
# refer to a mock endpoint name by that encrypted password
cool.result=mock:{{cool.password}}

# here is a password which is encrypted
cool.password=ENC(bsW9uV37gQ0QHFu7KO03Ww==)
```

Example with Spring XML

In Spring XML you need to configure the `JasyptPropertiesParser` which is shown below. Then the Camel Properties on page 387 component is told to use `jasypt` as the properties parser, which means Jasypt on page 253 have its chance to decrypt values looked up in the properties.

```
<!-- define the jasypt properties parser with the given password to be used -->
<bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <property name="password" value="secret"/>
</bean>

<!-- define the camel properties component -->
<bean id="properties" class="org.apache.camel.component.properties.PropertiesComponent">
    <!-- the properties file is in the classpath -->
    <property name="location" value="classpath:org/apache/camel/component/jasypt/myproper
ties.properties"/>
    <!-- and let it leverage the jasypt parser -->
```

```
    <property name="propertiesParser" ref="jasypt"/>
</bean>
```

The Properties on page 387 component can also be inlined inside the `<camelContext>` tag which is shown below. Notice how we use the `propertiesParserRef` attribute to refer to Jasypt on page 253.

```
<!-- define the jasypt properties parser with the given password to be used -->
<bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <!-- password is mandatory, you can prefix it with sysenv: or sys: to indicate it should
 use
        an OS environment or JVM system property value, so you dont have the master password
 defined here -->
    <property name="password" value="secret"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- define the camel properties placeholder, and let it leverage jasypt -->
    <propertyPlaceholder id="properties"
                         location="classpath:org/apache/camel/component/jasypt/myproper
ties.properties"
                         propertiesParserRef="jasypt"/>
    <route>
        <from uri="direct:start"/>
        <to uri="{{cool.result}}"/>
    </route>
</camelContext>
```

See Also

- Security
- Properties on page 387
- Encrypted passwords in ActiveMQ[3] - ActiveMQ has a similar feature as this `camel-jasypt` component

[3] http://activemq.apache.org/encrypted-passwords.html

Chapter 39. JBI

JBI Component

The **jbi** component is implemented by the ServiceMix Camel module[1] and provides integration with a JBI Normalized Message Router, such as the one provided by Apache ServiceMix[2].

Important

See below for information about how to use `StreamSource` types from ServiceMix[3] in Fuse Mediation Router.

The following code:

```
from("jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint")
```

Automatically exposes a new endpoint to the bus, where the service QName is `{http://foo.bar.org}MyService` and the endpoint name is `MyEndpoint` (see #URI-format).

When a JBI endpoint appears at the end of a route, for example:

```
to("jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint")
```

The messages sent by this producer endpoint are sent to the already deployed JBI endpoint.

URI format

```
jbi:service:serviceNamespace[sep]serviceName[?options]
jbi:endpoint:serviceNamespace[sep]serviceName[sep]endpointName[?options]
jbi:name:endpointName[?options]
```

The separator that should be used in the endpoint URL is:

- `/` (forward slash), if `serviceNamespace` starts with `http://`, or
- `:` (colon), if `serviceNamespace` starts with `urn:foo:bar`.

For more details of valid JBI URIs see the ServiceMix URI Guide[4].

[1] http://servicemix.apache.org/servicemix-camel.html
[2] http://servicemix.apache.org/
[3] http://servicemix.apache.org/
[4] http://servicemix.apache.org/uris.html

Using the `jbi:service:` or `jbi:endpoint:` URI formats sets the service QName on the JBI endpoint to the one specified. Otherwise, the default Fuse Mediation Router JBI Service QName is used, which is:

```
{http://activemq.apache.org/camel/schema/jbi}endpoint
```

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

Examples

```
jbi:service:http://foo.bar.org/MyService
jbi:endpoint:urn:foo:bar:MyService:MyEndpoint
jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint
jbi:name:cheese
```

URI options

Name	Default value	Description
`mep`	MEP of the Camel Exchange	Allows users to override the MEP set on the Exchange object. Valid values for this option are `in-only`, `in-out`, `robust-in-out` and `in-optional-out`.
`operation`	Value of the `jbi.operation` header property	Specifies the JBI operation for the `MessageExchange`. If no value is supplied, the JBI binding will use the value of the `jbi.operation` header property.
`serialization`	`basic`	Default value (`basic`) will check if headers are serializable by looking at the type, setting this option to `strict` will detect objects that can not be serialized although they implement the `Serializable` interface. Set to `nocheck` to disable this check altogether, note that this should only be used for in-memory transports like SEDAFlow, otherwise you can expect to get `NotSerializableException` thrown at runtime.
`convertException`	`false`	`false`: send any exceptions thrown from the Camel route back unmodified `true`: convert all exceptions to a JBI FaultException (can be used to avoid non-serializable exceptions or to implement generic error handling

Examples

```
jbi:service:http://foo.bar.org/MyService?mep=in-out       (override the MEP, use InOut JBI
 MessageExchanges)
jbi:endpoint:urn:foo:bar:MyService:MyEndpoint?mep=in     (override the MEP, use InOnly JBI
 MessageExchanges)
```

```
jbi:endpoint:urn:foo:bar:MyService:MyEndpoint?operation={http://www.mycompany.org}AddNumbers
  (overide the operation for the JBI Exchange to {http://www.mycompany.org}AddNumbers)
```

Using Stream bodies

If you are using a stream type as the message body, you should be aware that a stream is only capable of being read once. So if you enable `DEBUG` logging, the body is usually logged and thus read. To deal with this, Fuse Mediation Router has a `streamCaching` option that can cache the stream, enabling you to read it multiple times.

```
from("jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint").streamCaching().to("xslt:trans
form.xsl", "bean:doSomething");
```

From **Fuse Mediation Router 1.5** onwards, the stream caching is default enabled, so it is not necessary to set the `streamCaching()` option.

In **Fuse Mediation Router 2.0** we store big input streams (by default, over 64K) in a `temp` file using `CachedOutputStream`. When you close the input stream, the temp file will be deleted.

Creating a JBI Service Unit

If you have some Fuse Mediation Router routes that you want to deploy inside JBI as a Service Unit, you can use the JBI Service Unit Archetype to create a new Maven project for the Service Unit.

If you have an existing Maven project that you need to convert into a JBI Service Unit, you may want to consult ServiceMix Maven JBI Plugins[5] for further help. The key steps are as follows:

- Create a Spring XML file at `src/main/resources/camel-context.xml` to bootstrap your routes inside the JBI Service Unit.
- Change the POM file's packaging to `jbi-service-unit`.

Your `pom.xml` should look something like this to enable the `jbi-service-unit` packaging:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLS
chema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>myGroupId</groupId>
  <artifactId>myArtifactId</artifactId>
  <packaging>jbi-service-unit</packaging>
```

[5] http://servicemix.apache.org/maven-jbi-plugin.html

```
  <version>1.0-SNAPSHOT</version>

  <name>A Fuse Mediation Router based JBI Service Unit</name>

  <url>http://www.myorganization.org</url>

  <properties>
    <camel-version>1.0.0</camel-version>
    <servicemix-version>3.3</servicemix-version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.servicemix</groupId>
      <artifactId>servicemix-camel</artifactId>
      <version>${servicemix-version}</version>
    </dependency>

    <dependency>
      <groupId>org.apache.servicemix</groupId>
      <artifactId>servicemix-core</artifactId>
      <version>${servicemix-version}</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <defaultGoal>install</defaultGoal>

    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>

      <!-- creates the JBI deployment unit -->
      <plugin>
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <version>${servicemix-version}</version>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```

For more information, see the following references:

- ServiceMix Camel module[6]
- Using Camel with ServiceMix[7]
- Cookbook on using Camel with ServiceMix[8]

[6] http://servicemix.apache.org/servicemix-camel.html
[7] http://servicemix.apache.org/3-beginner-using-apache-camel-inside-servicemix.html
[8] http://servicemix.apache.org/order-file-processing.html

Chapter 40. JCR

JCR Component

The **`jcr`** component allows you to add nodes to a JCR (JSR-170) compliant content repository (for example, Apache Jackrabbit[1]).

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

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

URI format

```
jcr://user:password@repository/path/to/node
```

Usage

The `repository` element of the URI is used to look up the JCR `Repository` object in the Camel context registry.

If a message is sent to a JCR producer endpoint:

- A new node is created in the content repository,
- All the message properties of the IN message are transformed to JCR `Value` instances and added to the new node,
- The node's UUID is returned in the OUT message.

Message properties

All message properties are converted to node properties, except for the `CamelJcrNodeName` property (you can refer to `JcrConstants.NODE_NAME` in your code), which is used to determine the node name.

[1] http://jackrabbit.apache.org/

Example

The snippet below creates a node named `node` under the `/home/test` node in the content repository. One additional attribute is added to the node as well: `my.contents.property` which will contain the body of the message being sent.

```
from("direct:a").setProperty(JcrConstants.JCR_NODE_NAME, constant("node"))
    .setProperty("my.contents.property", body()).to("jcr://user:pass@repository/home/test");
```

Chapter 41. JDBC

JDBC Component

The **jdbc** component enables you to access databases through JDBC, where SQL queries and operations are sent in the message body. This component uses the standard JDBC API, unlike the SQL Component on page 493 component, which uses spring-jdbc.

Warning

This component can only be used to define producer endpoints, which means that you cannot use the JDBC component in a `from()` statement.

URI format

```
jdbc:dataSourceName[?options]
```

This component only supports producer endpoints.

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

Options

Name	Default Value	Description
`readSize`	`0` / `2000`	The default maximum number of rows that can be read by a polling query. The default value is 2000 for Fuse Mediation Router 1.5.0 or older. In newer releases the default value is 0.
`statement.<xxx>`	`null`	**Fuse Mediation Router 2.1:** Sets additional options on the `java.sql.Statement` that is used behind the scenes to execute the queries. For instance, `statement.maxRows=10`. For detailed documentation, see the `java.sql.Statement` javadoc[1] documentation.

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Statement.html

`useJDBC4ColumnNameAndLabelSemantics true`	**Fuse Mediation Router 1.6.3/2.2:** Sets whether to use JDBC 4/3 column label/name semantics. You can use this option to turn it `false` in case you have issues with your JDBC driver to select data. This only applies when using `SQL SELECT` using aliases (e.g. `SQL SELECT id as identifier, name as given_name from persons`).

Result

The result is returned in the OUT body as an `ArrayList<HashMap<String, Object>>`. The `List` object contains the list of rows and the `Map` objects contain each row with the `String` key as the column name.

Note

This component fetches `ResultSetMetaData` to be able to return the column name as the key in the `Map`.

Message Headers

Header	Description
`CamelJdbcRowCount`	If the query is a `SELECT`, the row count is returned in this OUT header.
`CamelJdbcUpdateCount`	If the query is an `UPDATE`, the update count is returned in this OUT header.

Samples

In the following example, we fetch the rows from the customer table.

First we register our datasource in the Fuse Mediation Router registry as `testdb`:

```
JndiRegistry reg = super.createRegistry();
reg.bind("testdb", ds);
return reg;
```

Then we configure a route that routes to the JDBC component, so the SQL will be executed. Note how we refer to the `testdb` datasource that was bound in the previous step:

```
// lets add simple route
public void configure() throws Exception {
```

```
    from("direct:hello").to("jdbc:testdb?readSize=100");
}
```

Or you can create a `DataSource` in Spring like this:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
     <from uri="timer://kickoff?period=10000"/>
     <setBody>
       <constant>select * from customer</constant>
     </setBody>
     <to uri="jdbc:testdb"/>
     <to uri="mock:result"/>
  </route>
</camelContext>
<!-- Just add a demo to show how to bind a date source for camel in Spring-->
<bean id="testdb" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
 <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
 <property name="url" value="jdbc:hsqldb:mem:camel_jdbc" />
 <property name="username" value="sa" />
  <property name="password" value="" />
</bean>
```

We create an endpoint, add the SQL query to the body of the IN message, and then send the exchange. The result of the query is returned in the OUT body:

```
// first we create our exchange using the endpoint
Endpoint endpoint = context.getEndpoint("direct:hello");
Exchange exchange = endpoint.createExchange();
// then we set the SQL on the in body
exchange.getIn().setBody("select * from customer order by ID");

// now we send the exchange to the endpoint, and receives the response from Camel
Exchange out = template.send(endpoint, exchange);

// assertions of the response
assertNotNull(out);
assertNotNull(out.getOut());
ArrayList<HashMap<String, Object>> data = out.getOut().getBody(ArrayList.class);
assertNotNull("out body could not be converted to an ArrayList - was: "
    + out.getOut().getBody(), data);
assertEquals(2, data.size());
HashMap<String, Object> row = data.get(0);
assertEquals("cust1", row.get("ID"));
assertEquals("jstrachan", row.get("NAME"));
row = data.get(1);
assertEquals("cust2", row.get("ID"));
assertEquals("nsandhu", row.get("NAME"));
```

If you want to work on the rows one by one instead of the entire ResultSet at once you need to use the Splitter EIP such as:

```
from("direct:hello")
        // here we split the data from the testdb into new messages one by one
        // so the mock endpoint will receive a message per row in the table
    .to("jdbc:testdb").split(body()).to("mock:result");
```

Sample - Polling the database every minute

If we want to poll a database using the JDBC component, we need to combine it with a polling scheduler such as the Timer on page 507 or Quartz on page 389 etc. In the following example, we retrieve data from the database every 60 seconds:

```
from("timer://foo?period=60000").setBody(constant("select * from customer")).to("jd
bc:testdb").to("activemq:queue:customers");
```

See also:

- SQL

Chapter 42. JDBC-AggregationRepository

JDBC-AggregationRepository

Available as of Camel 2.6

The `camel-jdbc-aggregator` component allows together with Camel to provide persistent support for the Aggregator.

Using JdbcAggregationRepository

`JdbcAggregationRepository` is an `AggregationRepository` which on the fly persists the aggregated messages. This ensures that you will not loose messages, as the default aggregator will use an in memory only `AggregationRepository`.

It has the following options:

Option	Type	Description
`dataSource`	`DataSource`	**Mandatory:** The `javax.sql.DataSource` to use for accessing the database.
`repositoryName`	`String`	**Mandatory:** The name of the repository.
`transactionManager`	`TransactionManager`	**Mandatory:** The `org.springframework.transaction.PlatformTransactionManager` to mange transactions for the database. The TransactionManager must be able to support databases.
`lobHandler`	`LobHandler`	A `org.springframework.jdbc.support.lob.LobHandler` to handle Lob types in the database. Use this option to use a vendor specific LobHandler, for example when using Oracle.
`returnOldExchange`	boolean	Whether the get operation should return the old existing Exchange if any existed. By default this option is `false` to optimize as we do not need the old exchange when aggregating.
`useRecovery`	boolean	Whether or not recovery is enabled. This option is by default `true`. When enabled the Camel Aggregator automatic recover failed aggregated exchange and have them resubmitted.
`recoveryInterval`	long	If recovery is enabled then a background task is run every x'th time to scan for failed exchanges to recover and resubmit. By default this interval is 5000 millis.
`maximumRedeliveries`	int	Allows you to limit the maximum number of redelivery attempts for a recovered exchange. If enabled then the Exchange will be moved to the

		dead letter channel if all redelivery attempts failed. By default this option is disabled. If this option is used then the `deadLetterUri` option must also be provided.
`deadLetterUri`	String	An endpoint uri for a Dead Letter Channel where exhausted recovered Exchanges will be moved. If this option is used then the `maximumRedeliveries` option must also be provided.

What is preserved when persisting

`JdbcAggregationRepository` will only preserve any `Serializable` compatible data types. If a data type is not such a type its dropped and a `WARN` is logged. And it only persists the `Message` body and the `Message` headers. The `Exchange` properties are **not** persisted.

Recovery

The `JdbcAggregationRepository` will by default recover any failed Exchange. It does this by having a background tasks that scans for failed Exchanges in the persistent store. You can use the `checkInterval` option to set how often this task runs. The recovery works as transactional which ensures that Camel will try to recover and redeliver the failed Exchange. Any Exchange which was found to be recovered will be restored from the persistent store and resubmitted and send out again.

The following headers is set when an Exchange is being recovered/redelivered:

Header	Type	Description
`Exchange.REDELIVERED`	Boolean	Is set to true to indicate the Exchange is being redelivered.
`Exchange.REDELIVERY_COUNTER`	Integer	The redelivery attempt, starting from 1.

Only when an Exchange has been successfully processed it will be marked as complete which happens when the `confirm` method is invoked on the `AggregationRepository`. This means if the same Exchange fails again it will be kept retried until it success.

You can use option `maximumRedeliveries` to limit the maximum number of redelivery attempts for a given recovered Exchange. You must also set the `deadLetterUri` option so Camel knows where to send the Exchange when the `maximumRedeliveries` was hit.

You can see some examples in the unit tests of camel-jdbc-aggregagor, for example this test[1].

[1] https://svn.apache.org/repos/asf/camel/trunk/components/camel-jdbc-aggregator/src/test/java/org/apache/camel/component/jdbc/aggregationrepository/JdbcAggregateRecoverDeadLetterChannelTest.java

Database

To be operational, each aggregator uses two table: the aggregation and completed one. By convention the completed has the same name as the aggregation one suffixed with `"_COMPLETED"`. The name must be configured in the Spring bean with the `RepositoryName` property. In the following example aggregation will be used.

The table structure definition of both table are identical: in both case a String value is used as key (**id**) whereas a Blob contains the exchange serialized in byte array. However one difference should be remembered: the **id** field does not have the same content depending on the table. In the aggregation table **id** holds the correlation Id used by the component to aggregate the messages. In the completed table, **id** holds the id of the exchange stored in corresponding the blob field.

Here is the SQL query used to create the tables, just replace `"aggregation"` with your aggregator repository name.

```
CREATE TABLE aggregation (
    id varchar(255) NOT NULL,
    exchange blob NOT NULL,
    constraint aggregation_pk PRIMARY KEY (id)
);
CREATE TABLE aggregation_completed (
    id varchar(255) NOT NULL,
    exchange blob NOT NULL,
    constraint aggregation_completed_pk PRIMARY KEY (id)
);
```

Codec (Serialization)

Since they can contain any type of payload, Exchanges are not serializable by design. It is converted into a byte array to be stored in a database BLOB field. All those conversions are handled by the `JdbcCodec` class. One detail of the code requires your attention: the `ClassLoadingAwareObjectInputStream`.

The `ClassLoadingAwareObjectInputStream` has been reused from the Apache ActiveMQ[2] project. It wraps an `ObjectInputStream` and use it with the `ContextClassLoader` rather than the `currentThread` one. The benefit is to be able to load classes exposed by other bundles. This allows the exchange body and headers to have custom types object references.

[2] http://activemq.apache.org/

Transaction

A Spring `PlatformTransactionManager` is required to orchestrate transaction.

Service (Start/Stop)

The `start` method verify the connection of the database and the presence of the required tables. If anything is wrong it will fail during starting.

Aggregator configuration

Depending on the targeted environment, the aggregator might need some configuration. As you already know, each aggregator should have its own repository (with the corresponding pair of table created in the database) and a data source. If the default lobHandler is not adapted to your database system, it can be injected with the `lobHandler` property.

Here is the declaration for Oracle:

```
    <bean id="lobHandler" class="org.springframework.jdbc.support.lob.OracleLobHandler">
        <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/>
    </bean>

    <bean id="nativeJdbcExtractor" class="org.springframework.jdbc.support.nativejdbc.Com
monsDbcpNativeJdbcExtractor"/>

   <bean id="repo" class="org.apache.camel.component.jdbc.aggregationRepository.JdbcAggreg
ationRepository">
        <property name="transactionManager" ref="transactionManager"/>
        <property name="repositoryName" value="aggregation"/>
        <property name="dataSource" ref="dataSource"/>
        <!-- Only with Oracle, else use default -->
        <property name="lobHandler" ref="lobHandler"/>
    </bean>
```

Dependencies

To use JDBC-AggregationRepository on page 271 in your camel routes you need to add the a dependency on **camel-jdbc-aggregator**.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions[3]).

[3] Download

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jdbc-aggregator</artifactId>
  <version>2.6.0</version>
</dependency>
```

- Aggregator
- HawtDB on page 209
- Components on page 3

Chapter 43. Jetty

Jetty Component

The **jetty** component provides HTTP-based endpoints[1] for consuming HTTP requests. That is, the Jetty component behaves as a simple Web server.

Upgrading from Jetty 6 to 7

You can read more about upgrading Jetty here[2]

URI format

```
jetty:http://hostname[:port][/resourceUri][?options]
```

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

Options

Name	Default Value	Description
`sessionSupport`	`false`	Specifies whether to enable the session manager on the server side of Jetty.
`httpClient.XXX`	`null`	**Fuse Mediation Router 1.6.0/2.0:** Configuration of Jetty's HttpClient[3]. For example, setting `httpClient.idleTimeout=30000` sets the idle timeout to 30 seconds.
`httpBindingRef`	`null`	**Fuse Mediation Router 1.6.0/2.0:** Reference to an `org.apache.camel.component.http.HttpBinding` in the Registry. `HttpBinding` can be used to customize how a response should be written.
`jettyHttpBindingRef`	`null`	**Camel 2.6.0:** Reference to an `org.apache.camel.component.jetty.JettyHttpBinding` in the Registry. `JettyHttpBinding` can be used to customize how a response should be written.

[1] Endpoint
[2] http://wiki.eclipse.org/Jetty/Howto/Upgrade_from_Jetty_6_to_Jetty_7
[3] http://wiki.eclipse.org/Jetty/Tutorial/HttpClient

`matchOnUriPrefix`	`false`	**Fuse Mediation Router 2.0:** Whether or not the `CamelServlet` should try to find a target consumer by matching the URI prefix if no exact match is found.
`handlers`	`null`	**Fuse Mediation Router 1.6.1/2.0:** Specifies a comma-delimited set of `org.mortbay.jetty.Handler` instances in your Registry (such as your Spring `ApplicationContext`). These handlers are added to the Jetty servlet context (for example, to add security).
`chunked`	`true`	**Camel 2.2:** If this option is false Jetty servlet will disable the HTTP streaming and set the content-length header on the response
`enableJmx`	`false`	**Camel 2.3:** If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.
`disableStreamCache`	`false`	**Camel 2.3:** Determines whether or not the raw input stream from Jetty is cached or not (Camel will read the stream into a in memory/overflow to file, Stream caching) cache. By default Camel will cache the Jetty input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to `true` when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is `false` to support reading the stream multiple times.
`bridgeEndpoint`	`false`	**Camel 2.1:** If the option is true , HttpProducer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for request. You may also set the **throwExcpetionOnFailure** to be false to let the HttpProducer send all the fault response back. **Camel 2.3:** If the option is true, HttpProducer and CamelServlet will skip the gzip processing if the content-encoding is "gzip".
`enableMultipartFilter`	`true`	**Canel 2.5:** Whether Jetty `org.eclipse.jetty.servlets.MultiPartFilter` is enabled or not. You should set this value to `false` when bridging endpoints, to ensure multipart requests is proxied/bridged as well.
`multipartFilterRef`	`null`	**Camel 2.6:** Allows using a custom multipart filter. Note: setting `multipartFilterRef` forces the value of `enableMultipartFilter` to `true`.
`continuationTimeout`	`null`	**Camel 2.6:** Allows to set a timeout in millis when using Jetty on page 277 as consumer (server). By default Jetty uses 30000. You can use a value of `<= 0` to never expire. If a timeout occurs then the request will be expired and Jetty will return back a http error 503 to the client. This option is only in use when using Jetty on page 277 with the Asynchronous Routing Engine.

`useContinuation`	`true`	**Camel 2.6:** Whether or not to use Jetty continuations[4] for the Jetty Server.

DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times. |

Message Headers

Fuse Mediation Router uses the same message headers as the HTTP on page 229 component. From Camel 2.2, it also uses (Exchange.HTTP_CHUNKED,CamelHttpChunked) header to turn on or turn off the chuched encoding on the camel-jetty consumer.

Fuse Mediation Router also populates **all** request.parameter and request.headers. For example, given a client request with the URL, `http://myserver/myserver?orderid=123`, the exchange will contain a header named `orderid` with the value 123. This feature was introduced in Fuse Mediation Router 1.5.

From Camel 1.6.3 and Camel 2.2.0, you can get the request.parameter from the message header not only from Get Method, but also other HTTP method.

Usage

The Jetty component only supports consumer endpoints. Therefore a Jetty endpoint URI should be used only as the **input** for a Fuse Mediation Router route (in a `from()` DSL call). To issue HTTP requests against other HTTP endpoints, use the HTTP Component on page 229

Component Options

The `JettyHttpComponent` provides the following options:

Name	Default Value	Description
`enableJmx`	`false`	**Camel 2.3:** If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.
`sslKeyPassword`	`null`	**Consumer only**: The password for the keystore when using SSL.
`sslPassword`	`null`	**Consumer only**: The password when using SSL.
`sslKeystore`	`null`	**Consumer only**: The path to the keystore.
`minThreads`	`null`	**Camel 2.5Consumer only**: To set a value for minimum number of threads in server thread pool.

[4] http://wiki.eclipse.org/Jetty/Feature/Continuations

`maxThreads`	`null`	**Camel 2.5Consumer only**: To set a value for maximum number of threads in server thread pool.
`threadPool`	`null`	**Camel 2.5Consumer only**: To use a custom thread pool for the server.
`sslSocketConnectors`	`null`	**Camel 2.3Consumer only:** A map which contains per port number specific SSL connectors. See section *SSL support* for more details.
`socketConnectors`	`null`	**Camel 2.5Consumer only:** A map which contains per port number specific HTTP connectors. Uses the same principle as `sslSocketConnectors` and therefore see section *SSL support* for more details.
`sslSocketConnectorProperties`	`null`	**Camel 2.5Consumer only**. A map which contains general SSL connector properties. See section *SSL support* for more details.
`socketConnectorProperties`	`null`	**Camel 2.5Consumer only**. A map which contains general HTTP connector properties. Uses the same principle as `sslSocketConnectorProperties` and therefore see section *SSL support* for more details.
`httpClient`	`null`	**Producer only**: To use a custom `HttpClient` with the jetty producer.
`httpClientMinThreads`	`null`	**Producer only**: To set a value for minimum number of threads in `HttpClient` thread pool.
`httpClientMaxThreads`	`null`	**Producer only**: To set a value for maximum number of threads in `HttpClient` thread pool.
`httpClientThreadPool`	`null`	**Producer only**: To use a custom thread pool for the client.

Sample

In this sample we define a route that exposes a HTTP service at `http://localhost:8080/myapp/myservice`:

```
from("jetty:http://localhost:9080/myapp/myservice").process(new MyBookService());
```

Usage of localhost

When you specify `localhost` in a URL, Fuse Mediation Router exposes the endpoint only on the local TCP/IP network interface, so it cannot be accessed from outside the machine it operates on.

If you need to expose a Jetty endpoint on a specific network interface, the numerical IP address of this interface should be used as the host. If you need to expose a Jetty endpoint on all network interfaces, the `0.0.0.0` address should be used.

Our business logic is implemented in the `MyBookService` class, which accesses the HTTP request contents and then returns a response. **Note:** The `assert` call appears in this example, because the code is part of an unit test.

```
public class MyBookService implements Processor {
    public void process(Exchange exchange) throws Exception {
        // just get the body as a string
        String body = exchange.getIn().getBody(String.class);

        // we have access to the HttpServletRequest here and we can grab it if we need it
        HttpServletRequest req = exchange.getIn().getBody(HttpServletRequest.class);
        assertNotNull(req);

        // for unit testing
        assertEquals("bookid=123", body);

        // send a html response
       exchange.getOut().setBody("<html><body>Book 123 is Camel in Action</body></html>");

    }
}
```

The following sample shows a content-based route that routes all requests containing the URI parameter, `one`, to the endpoint, `mock:one`, and all others to `mock:other`.

```
from("jetty:" + serverUri)
    .choice()
    .when().simple("in.header.one").to("mock:one")
    .otherwise()
    .to("mock:other");
```

So if a client sends the HTTP request, `http://serverUri?one=hello`, the Jetty component will copy the HTTP request parameter, `one` to the exchange's `in.header`. We can then use the `simple` language to route exchanges that contain this header to a specific endpoint and all others to another. If we used a language more powerful than Simple—such as El or OGNL—we could also test for the parameter value and do routing based on the header value as well.

Session Support

The session support option, `sessionSupport`, can be used to enable a `HttpSession` object and access the session object while processing the exchange. For example, the following route enables sessions:

```
<route>
    <from uri="jetty:http://0.0.0.0/myapp/myservice/?sessionSupport=true"/>
    <processRef ref="myCode"/>
<route>
```

The `myCode` Processor can be instantiated by a Spring `bean` element:

```
   <bean id="myCode" class="com.mycompany.MyCodeProcessor"/>
```

Where the processor implementation can access the `HttpSession` as follows:

```
public void process(Exchange exchange) throws Exception {
    HttpSession session = ((HttpExchange)exchange).getRequest().getSession();
    ...
}
```

SSL Support (HTTPS)

Jetty provides SSL support out of the box. To enable Jetty to run in SSL mode, simply format the URI with the `https://` prefix—for example:

```
<from uri="jetty:https://0.0.0.0/myapp/myservice/"/>
```

Jetty also needs to know where to load your keystore from and what passwords to use in order to load the correct SSL certificate. Set the following JVM System Properties:

until Camel 2.2

- `jetty.ssl.keystore` specifies the location of the Java keystore file, which contains the Jetty server's own X.509 certificate in a *key entry*. A key entry stores the X.509 certificate (effectively, the *public key*) and also its associated private key.
- `jetty.ssl.password` the store password, which is required to access the keystore file (this is the same password that is supplied to the `keystore` command's `-storepass` option).
- `jetty.ssl.keypassword` the key password, which is used to access the certificate's key entry in the keystore (this is the same password that is supplied to the `keystore` command's `-keypass` option).

from Camel 2.3 onwards

- `org.eclipse.jetty.ssl.keystore` specifies the location of the Java keystore file, which contains the Jetty server's own X.509 certificate in a *key entry*. A key entry stores the X.509 certificate (effectively, the *public key*) and also its associated private key.
- `org.eclipse.jetty.ssl.password` the store password, which is required to access the keystore file (this is the same password that is supplied to the `keystore` command's `\-storepass` option).
- `org.eclipse.jetty.ssl.keypassword` the key password, which is used to access the certificate's key entry in the keystore (this is the same password that is supplied to the `keystore` command's `\-keypass` option).

For details of how to configure SSL on a Jetty endpoint, see How to Configure SSL[5].

Some SSL properties aren't exposed directly by Camel, however Camel does expose the underlying SslSocketConnector, which will allow you to set properties like needClientAuth for mutual authentication requiring a client certificate or wantClientAuth for mutual authentication where a client doesn't need a certificate but can have one. There's a slight difference between Camel 1.6.x and 2.x:

Camel 1.x

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
  <property name="sslSocketConnector">
    <bean class="org.mortbay.jetty.security.SslSocketConnector">
      <property name="password" value="..." />
      <property name="keyPassword" value="..." />
      <property name="keystore" value="..." />
      <property name="wantClientAuth" value="..." />
      <property name="truststore" value="..." />
    </bean>
  </property>
</bean>
```

Until Camel 2.2

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
   <property name="sslSocketConnectors">
      <map>
         <entry key="8043">
           <bean class="org.mortbay.jetty.security.SslSocketConnector">
             <property name="password" value="..." />
             <property name="keyPassword" value="..." />
             <property name="keystore" value="..." />
             <property name="needClientAuth" value="..." />
```

[5] http://docs.codehaus.org/display/JETTY/How+to+configure+SSL

```
              <property name="truststore" value="..." />
            </bean>
          </entry>
        </map>
    </property>
  </bean>
```

Camel 2.3 to 2.4

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
    <property name="sslSocketConnectors">
        <map>
            <entry key="8043">
                <bean class="org.eclipse.jetty.server.ssl.SslSocketConnector">
                    <property name="password"value="..."/>
                    <property name="keyPassword"value="..."/>
                    <property name="keystore"value="..."/>
                    <property name="needClientAuth"value="..."/>
                    <property name="truststore"value="..."/>
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

From Camel 2.5 we switch to use `SslSelectChannelConnector`

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
    <property name="sslSocketConnectors">
        <map>
            <entry key="8043">
                <bean class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
                    <property name="password"value="..."/>
                    <property name="keyPassword"value="..."/>
                    <property name="keystore"value="..."/>
                    <property name="needClientAuth"value="..."/>
                    <property name="truststore"value="..."/>
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

The value you use as keys in the above map is the port you configure Jetty to listen on.

Configuring general SSL properties

Available as of Camel 2.5

Instead of a per port number specific SSL socket connector (as shown above) you can now configure general properties which applies for all SSL socket connectors (which is not explicit configured as above with the port number as entry).

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
    <property name="sslSocketConnectorProperties">
        <properties>
            <property name="password"value="..."/>
            <property name="keyPassword"value="..."/>
            <property name="keystore"value="..."/>
            <property name="needClientAuth"value="..."/>
            <property name="truststore"value="..."/>
        </properties>
    </property>
</bean>
```

Configuring general HTTP properties

Available as of Camel 2.5

Instead of a per port number specific HTTP socket connector (as shown above) you can now configure general properties which applies for all HTTP socket connectors (which is not explicit configured as above with the port number as entry).

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
    <property name="socketConnectorProperties">
        <properties>
            <property name="acceptors" value="4"/>
            <property name="maxIdleTime" value="300000"/>
        </properties>
    </property>
</bean>
```

Default behavior for returning HTTP status codes

The default behavior of HTTP status codes is defined by the `org.apache.camel.component.http.DefaultHttpBinding` class, which handles how a response is written and also sets the HTTP status code.

If the exchange was processed successfully, the 200 HTTP status code is returned. If the exchange failed with an exception, the 500 HTTP status code is returned, and the stacktrace is returned in the body. If you want to

specify which HTTP status code to return, set the code in the `HttpProducer.HTTP_RESPONSE_CODE` header of the OUT message.

Customizing HttpBinding

Available as of Fuse Mediation Router 1.5.1/2.0

By default, Fuse Mediation Router uses the `org.apache.camel.component.http.DefaultHttpBinding` to handle how a response is written. If you like, you can customize this behavior either by implementing your own `HttpBinding` class or by extending `DefaultHttpBinding` and overriding the appropriate methods.

The following example shows how to customize the `DefaultHttpBinding` in order to change how exceptions are returned:

```
public class MyJettyHttpBinding extends DefaultJettyHttpBinding {
    @Override
    protected void populateResponse(Exchange exchange, JettyContentExchange httpExchange,
Message in,
                                    HeaderFilterStrategy strategy, int responseCode) throws
 IOException {

        Message answer = exchange.getOut();

        answer.setHeaders(in.getHeaders());
        answer.setHeader(Exchange.HTTP_RESPONSE_CODE, responseCode);
        answer.setBody("Not exactly the message the server returned.");
    }
}
```

We can then create an instance of our binding and register it in the Spring registry as follows:

```
   <bean id="mybinding" class="com.mycompany.MyHttpBinding"/>
```

And then we can reference this binding when we define the route:

```
   <route>
     <from uri="jetty:http://0.0.0.0:8080/myapp/myservice?httpBindingRef=mybinding"/>
     <to uri="bean:doSomething"/>
   </route>
```

Jetty handlers and security configuration

Available as of Fuse Mediation Router 1.6.1/2.0: You can configure a list of Jetty handlers on the endpoint, which can be useful for enabling advanced Jetty security features. These handlers are configured in Spring XML as follows:

```
       <-- Jetty Security handling -->
 <bean id="userRealm" class="org.mortbay.jetty.plus.jaas.JAASUserRealm">
  <property name="name" value="tracker-users" />
  <property name="loginModuleName" value="ldaploginmodule" />
 </bean>

 <bean id="constraint" class="org.mortbay.jetty.security.Constraint">
  <property name="name" value="BASIC" />
  <property name="roles" value="tracker-users" />
  <property name="authenticate" value="true" />
 </bean>

 <bean id="constraintMapping" class="org.mortbay.jetty.security.ConstraintMapping">
  <property name="constraint" ref="constraint" />
  <property name="pathSpec" value="/*" />
 </bean>

 <bean id="securityHandler" class="org.mortbay.jetty.security.SecurityHandler">
  <property name="userRealm" ref="userRealm" />
  <property name="constraintMappings" ref="constraintMapping"/></bean>
```

And from Camel 2.3 onwards you can configure a list of Jetty handlers as follows:

```
<-- Jetty Security handling -->
<bean id="constraint" class="org.eclipse.jetty.http.security.Constraint">
    <property name="name" value="BASIC"/>
    <property name="roles" value="tracker-users"/>
    <property name="authenticate" value="true"/>
</bean>

<bean id="constraintMapping" class="org.eclipse.jetty.security.ConstraintMapping">
    <property name="constraint" ref="constraint"/>
    <property name="pathSpec" value="/*"/>
</bean>

<bean id="securityHandler" class="org.eclipse.jetty.security.ConstraintSecurityHandler">
    <property name="authenticator">
        <bean class="org.eclipse.jetty.security.authentication.BasicAuthenticator"/>
    </property>
    <property name="constraintMappings">
        <list>
            <ref bean="constraintMapping"/>
        </list>
    </property>
</bean>
```

You can then define the endpoint as:

```
from("jetty:http://0.0.0.0:9080/myservice?handlers=securityHandler")
```

If you need more handlers, set the `handlers` option equal to a comma-separated list of bean IDs.

How to return a custom HTTP 500 reply message

You may want to return a custom reply message when something goes wrong, instead of the default reply message Camel Jetty on page 277 replies with. You could use a custom `HttpBinding` to be in control of the message mapping, but often it may be easier to use Camel's Exception Clause to construct the custom reply message. For example as show here, where we return `Dude something went wrong` with HTTP error code 500:

```
from("jetty://http://localhost:8234/myserver")
    // use onException to catch all exceptions and return a custom reply message
    .onException(Exception.class)
        .handled(true)
        // create a custom failure response
        .transform(constant("Dude something went wrong"))
        // we must remember to set error code 500 as handled(true)
        // otherwise would let Camel thing its a OK response (200)
        .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(500))
    .end()
    // now just force an exception immediately
    .throwException(new IllegalArgumentException("I cannot do this"));
```

Multi-part Form support

From Camel 2.3.0, camel-jetty support to multipart form post out of box. The submitted form-data are mapped into the message header. Camel-jetty creates an attachment for each uploaded file. The file name is mapped to the name of the attachment. The content type is set as the content type of the attachment file name. You can find the example here.

```
// Set the jetty temp directory which store the file for multi part form
// camel-jetty will clean up the file after it handled the request.
// The option works rightly from Camel 2.4.0
getContext().getProperties().put("CamelJettyTempDir", "target");

from("jetty://http://localhost:9080/test").process(new Processor() {

    public void process(Exchange exchange) throws Exception {
        Message in = exchange.getIn();
        assertEquals("Get a wrong attachement size", 1, in.getAttachments().size());
        // The file name is attachment id
        DataHandler data = in.getAttachment("NOTICE.txt");
```

```
        assertNotNull("Should get the DataHandle NOTICE.txt", data);
        assertEquals("Get a wrong content type", "text/plain", data.getContentType());
        assertEquals("Got the wrong name", "NOTICE.txt", data.getName());

        assertTrue("We should get the data from the DataHandle", data.getDataSource()
            .getInputStream().available() > 0);

        // The other form date can be get from the message header
        exchange.getOut().setBody(in.getHeader("comment"));
    }

});
```

Jetty JMX support

From Camel 2.3.0, camel-jetty supports the enabling of Jetty's JMX capabilities at the component and endpoint level with the endpoint configuration taking priority. Note that JMX must be enabled within the Camel context in order to enable JMX support in this component as the component provides Jetty with a reference to the MBeanServer registered with the Camel context. Because the camel-jetty component caches and reuses Jetty resources for a given protocol/host/port pairing, this configuration option will only be evaluated during the creation of the first endpoint to use a protocol/host/port pairing. For example, given two routes created from the following XML fragments, JMX support would remain enabled for all endpoints listening on "https://0.0.0.0".

```
<from uri="jetty:https://0.0.0.0/myapp/myservice1/?enableJmx=true"/>
```

```
<from uri="jetty:https://0.0.0.0/myapp/myservice2/?enableJmx=false"/>
```

The camel-jetty component also provides for direct configuration of the Jetty MBeanContainer. Jetty creates MBean names dynamically. If you are running another instance of Jetty outside of the Camel context and sharing the same MBeanServer between the instances, you can provide both instances with a reference to the same MBeanContainer in order to avoid name collisions when registering Jetty MBeans.

See also:

- Http

Chapter 44. Jing

Jing Component

The Jing component uses the Jing Library[1] to perform XML validation of the message body using either:

- RelaxNG XML Syntax[2]
- RelaxNG Compact Syntax[3]

Note that the MSV on page 361 component can also support RelaxNG XML syntax.

URI format

```
rng:someLocalOrRemoteResource
rnc:someLocalOrRemoteResource
```

Where **rng** means use the RelaxNG XML Syntax[4] whereas **rnc** means use RelaxNG Compact Syntax[5]. The following examples show possible URI values

Example	Description
`rng:foo/bar.rng`	References the XML file **foo/bar.rng** on the classpath
`rnc:http://foo.com/bar.rnc`	References the RelaxNG Compact Syntax file from the URL, `http://foo.com/bar.rnc`.

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

Options

Option	Default	Description
`useDom`	`false`	**Fuse Mediation Router 2.0:** Specifies whether DOMSource/DOMResult or SaxSource/SaxResult should be used by the validator.

[1] http://www.thaiopensource.com/relaxng/jing.html
[2] http://relaxng.org/
[3] http://relaxng.org/compact-tutorial-20030326.html
[4] http://relaxng.org/
[5] http://relaxng.org/compact-tutorial-20030326.html

Example

The following example[6] shows how to configure a route from the endpoint **direct:start** which then goes to one of two endpoints, either **mock:valid** or **mock:invalid** based on whether or not the XML matches the given RelaxNG Compact Syntax[7] schema (which is supplied on the classpath).

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <doTry>
            <to uri="rnc:org/apache/camel/component/validator/jing/schema.rnc"/>
            <to uri="mock:valid"/>

            <doCatch>
                <exception>org.apache.camel.ValidationException</exception>
                <to uri="mock:invalid"/>
            </doCatch>
            <doFinally>
                <to uri="mock:finally"/>
            </doFinally>
        </doTry>
    </route>
</camelContext>
```

6 http://svn.apache.org/repos/asf/camel/trunk/components/camel-jing/src/test/resources/org/apache/camel/component/validator/jing/rnc-context.xml

7 http://relaxng.org/compact-tutorial-20030326.html

Chapter 45. JMS

JMS Component

The JMS component allows messages to be sent to (or consumed from) a JMS[1] 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.

For users with Fuse Mediation Router 1.6.1 or older

JMS consumers have a bad default in Fuse Mediation Router 1.6.1 or older. The `maxMessagesPerTask` is set to 1, whereas it really should be -1. This issue causes Spring to create a new thread after it has processed a message, causing the thread count to rise continuously. You can see this in the log where a new thread name is used. To remedy this, change a route such as:

```
<from uri="jms:queue:foo"/>
```

By adding the `maxMessagesPerTask` option and setting its value to -1, as follows:

```
<from uri="jms:queue:foo&axMessagesPerTask=-1"/>
```

This has been fixed in Fuse Mediation Router 1.6.2/2.0.

Using ActiveMQ

If you are using Apache ActiveMQ[2], you should prefer the ActiveMQ on page 25 component as it has been particularly optimized for ActiveMQ on page 25. All of the options and samples on this page are also valid for the ActiveMQ on page 25 component.

Using JMS API 1.0.2

The old JMS API 1.0.2 has been @**deprecated** in Camel 2.1 and will be removed in Camel 2.2 release. Its no longer provided in Spring 3.0 which we want to be able to support out of the box in Camel 2.2+ releases.

[1] http://java.sun.com/products/jms/
[2] http://activemq.apache.org/

URI format

```
jms:[temp:][queue:|topic:]destinationName[?options]
```

Where `destinationName` is a JMS queue or topic name. By default, the `destinationName` is interpreted as a queue name. For example, to connect to the queue, FOO.BAR, use:

```
jms:FOO.BAR
```

You can include the optional `queue:` prefix, if you prefer:

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

To connect to a topic, you *must* include the `topic:` prefix. For example, to connect to the topic, `Stocks.Prices`, use:

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

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

Using Temporary Destinations

As of Fuse Mediation Router 1.4.0, you can access temporary queues using the following URL format:

```
jms:temp:queue:foo
```

Or temporary topics using the following URL format:

```
jms:temp:topic:bar
```

This URL format enables multiple routes or processors or beans to refer to the same temporary destination. For example, you can create three temporary destinations and use them in routes as inputs or outputs by referring to them by name.

Notes

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 in the JMS provider to avoid performance being lousy[3]. So if you intend to use Apache ActiveMQ[4] as your Message Broker - which is a good choice as ActiveMQ rocks :-) , then we recommend that you either

- Use the ActiveMQ on page 25 component, which is already configured to use ActiveMQ efficiently, or
- Use the `PoolingConnectionFactory` in ActiveMQ.

If you wish to use durable topic subscriptions, you need to specify both **clientId** and **durableSubscriptionName**. 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[5] instead to avoid this limitation. More background on durable messaging here[6].

When using message headers, the JMS specification states that header names must be valid Java identifiers. So, by default, Fuse Mediation Router ignores any headers that do not match this rule. So try to name your headers as if they are valid Java identifiers. One benefit of doing this is that you can then use your headers inside a JMS Selector (whose SQL92 syntax mandates Java identifier syntax for headers).

From Fuse Mediation Router 1.4 onwards, a simple strategy for mapping header names is used by default. The strategy is to replace any dots in the header name with the underscore character and to reverse the replacement when the header name is restored from a JMS message sent over the wire. What does this mean? No more losing method names to invoke on a bean component, no more losing the filename header for the File Component, and so on.

The current header name strategy for accepting header names in Fuse Mediation Router is as follows:

- Replace all dots with underscores (for example, `org.apache.camel.MethodName` becomes `org_apache_camel_MethodName`).
- Test if the name is a valid java identifier using the JDK core classes.

[3] http://activemq.apache.org/jmstemplate-gotchas.html
[4] http://activemq.apache.org/
[5] http://activemq.apache.org/virtual-destinations.html
[6] http://activemq.apache.org/how-do-durable-queues-and-topics-work.html

- If the test success, the header is added and sent over the wire; otherwise it is dropped (and logged at `DEBUG` level).

In Fuse Mediation Router 2.0 the strategy for mapping header names has been changed to use the following replacement strategy:

- Dots are replaced by `_DOT_` and the replacement is reversed when Fuse Mediation Router consume the message
- Hyphen is replaced by `_HYPHEN_` and the replacement is reversed when Fuse Mediation Router consumes the message

For Consuming Messages cacheLevelName settings are vital!

If you are using Spring before 2.5.1 and Fuse Mediation Router before 1.3.0, you might want to set the `cacheLevelName` to be `CACHE_CONSUMER` for maximum performance. Due to a bug in earlier Spring versions[7] causing a lack of transactional integrity, previous versions of Fuse Mediation Router and Fuse Mediation Router versions from 1.3.0 onwards when used with Spring versions earlier than 2.5.1 will default to using `CACHE_CONNECTION`. See the JIRAs CAMEL-163[8] and CAMEL-294[9]. Also, if you are using XA resources or running in a J2EE container, you may want to set the `cacheLevelName` to be `CACHE_NONE` as we have found that when using JBoss with TibCo EMS and JTA/XA you must disable caching. Another user reports problems using WebSphere MQ 6.0.2.5, Fuse Mediation Router 1.6.0 and Spring 2.5.6. The application does not use XA and is not running inside a J2EE Container, but the `cacheLevelName=CACHE_NONE` setting seems to solve the problem with WebSphere MQ. See also more about JmsTemplate gotchas[10].

Options

You can configure many different properties on the JMS endpoint which map to properties on the JMSConfiguration POJO[11]. **Note:** Many of these properties map to properties on Spring JMS, which Fuse Mediation Router uses for sending and receiving messages. So you can get more information about these properties by consulting the relevant Spring documentation.

Option	Default Value	Description
`acceptMessagesWhileStopping`	`false`	Specifies whether the consumer accept messages while it is sto

[7] http://opensource.atlassian.com/projects/spring/browse/SPR-3890
[8] https://issues.apache.org/activemq/browse/CAMEL-163
[9] https://issues.apache.org/activemq/browse/CAMEL-294
[10] http://activemq.apache.org/jmstemplate-gotchas.html
[11] http://camel.apache.org/maven/current/camel-jms/apidocs/org/apache/camel/component/jms/JmsConfiguration.html

acknowledgementModeName	AUTO_ACKNOWLEDGE	The JMS acknowledgement name, which is one of: TRANSAC AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE
acknowledgementMode	-1	The JMS acknowledgement mode defined as an Integer. Allo the acknowledgment mode. For the regular modes, it is prefer instead.
alwaysCopyMessage	false	If true, Fuse Mediation Router will always make a JMS mess to the producer for sending. Copying the message is needed replyToDestinationSelectorName is set (incidentally, Fuse alwaysCopyMessage option to true, if a replyToDestinatio
autoStartup	true	Specifies whether the consumer container should auto-startu
cacheLevelName	CACHE_CONSUMER	Sets the cache level by name for the underlying JMS resourc CACHE_CONNECTION, CACHE_CONSUMER, CACHE_NONE, and CACHE and see the warning above.
cacheLevel	-1	Sets the cache level by ID for the underlying JMS resources.
clientId	null	Sets the JMS client ID to use. Note that this value, if specifie a single JMS connection instance. It is typically only required prefer to use Virtual Topics[13] instead.
consumerType	Default	The consumer type to use, which can be one of: Simple, Defa type determines which Spring JMS listener to use. Default v org.springframework.jms.listener.DefaultMessageLis org.springframework.jms.listener.SimpleMessageList will use org.springframework.jms.listener.serversession.Ser If the option, useVersion102=true, Fuse Mediation Router wi ServerSessionPool is **@deprecated** and will be removed in
concurrentConsumers	1	Specifies the default number of concurrent consumers.
connectionFactory	null	The default JMS connection factory to use for the listenerC templateConnectionFactory, if neither is specified.
deliveryMode	2	Specifies the delivery mode when sending, where 1 = non-pe
deliveryPersistent	true	Specifies whether persistent delivery is used by default.
destination	null	**Fuse Mediation Router 2.0:** Specifies the JMS Destination
destinationName	null	**Fuse Mediation Router 2.0:** Specifies the JMS destination n
destinationResolver	null	A pluggable org.springframework.jms.support.destina you to use your own resolver (for example, to lookup the real

[12] http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/jms/listener/DefaultMessageListenerContainer.html
[13] http://activemq.apache.org/virtual-destinations.html

disableReplyTo	false	If true, ignore the JMSReplyTo header and so treat messages a reply back.
durableSubscriptionName	null	The durable subscriber name for specifying durable topic subsc configured as well.
eagerLoadingOfProperties	false	Enables eager loading of JMS properties as soon as a message i because the JMS properties might not be required. But this featu with the underlying JMS provider and the use of JMS properties. purposes, to ensure JMS properties can be understood and han
exceptionListener	null	Specifies the JMS Exception Listener that is to be notified of an
explicitQosEnabled	false	Set if the deliveryMode, priority or timeToLive qualities of s messages. This option is based on Spring's JmsTemplate. The de options are applied to the current endpoint. This contrasts with t operates at message granularity, reading QoS properties exclus message headers.
exposeListenerSession	true	Specifies whether the listener session should be exposed when
idleTaskExecutionLimit	1	Specifies the limit for idle executions of a receive task, not havin execution. If this limit is reached, the task will shut down and lea the case of dynamic scheduling; see the maxConcurrentConsum
jmsMessageType	null	**Fuse Mediation Router 2.0:** Allows you to force the use of a spec for sending JMS messages. Possible values are: Bytes, Map, Ob Mediation Router would determine which JMS message type to allows you to specify it.
jmsKeyFormatStrategy	default	**Fuse Mediation Router 2.0:** Pluggable strategy for encoding an compliant with the JMS specification. Fuse Mediation Router pro box: default and passthrough. The default strategy will safe The passthrough strategy leaves the key as is. Can be used for JMS header keys contain illegal characters. You can provide yo org.apache.camel.component.jms.JmsKeyFormatStrategy a
jmsOperations	null	Allows you to use your own implementation of the org.springf interface. Fuse Mediation Router uses JmsTemplate as default. used much as stated in the spring API docs.
lazyCreateTransactionManager	true	**Fuse Mediation Router 2.0:** If true, Fuse Mediation Router wil there is no transactionManager injected when option transac
listenerConnectionFactory	null	The JMS connection factory used for consuming messages.
mapJmsMessage	true	**Fuse Mediation Router 1.6.2/2.0:** Specifies whether Fuse Mediat JMS message to an appropiate payload type, such as javax.jm section about how mapping works below for more details.
maxConcurrentConsumers	1	Specifies the maximum number of concurrent consumers.

maxMessagesPerTask	-1	The number of messages per task. -1 is unlimited.
messageConverter	null	**Fuse Mediation Router 1.6.2/2.0:** To use a custom Spring org.springframework.jms.support.converter.MessageC how to map to/from a javax.jms.Message.
messageIdEnabled	true	When sending, specifies whether message IDs should be add the UUID generator registered with the CamelContext—for d *Programming EIP Components*.
messageTimestampEnabled	true	Specifies whether timestamps should be enabled by default
password	null	The password for the connector factory.
priority	4	Values greater than 1 specify the message priority when sen the highest). The explicitQosEnabled option **must** also be effect.
pubSubNoLocal	false	Specifies whether to inhibit the delivery of messages publish
receiveTimeout	*None*	The timeout for receiving messages (in milliseconds).
recoveryInterval	5000	Specifies the interval between recovery attempts, in millisec seconds.
preserveMessageQos	false	**Camel 2.0**: Set to true, if you want to send message using t instead of the QoS settings on the JMS endpoint. The following JMSDeliveryMode, and JMSExpiration. You can provide all will fall back to use the values from the endpoint instead. So, v the values from the endpoint. The explicitQosEnabled opti the endpoint, and not values from the message header.
replyTo	null	Provides an explicit ReployTo destination, which overrides ar Message.getJMSReplyTo().
replyToDestinationSelectorName	null	Sets the JMS Selector using the fixed name to be used so yo others when using a shared queue (that is, if you are not usi
replyToDeliveryPersistent	true	Specifies whether to use persistent delivery by default for rep
requestTimeout	20000	The timeout for waiting for a reply when using the InOut Exch is 20 seconds.
selector	null	Sets the JMS Selector, which is an SQL 92 predicate that is You may have to encode special characters such as = as %3
subscriptionDurable	false	@**deprecated:** Enabled by default, if you specify a durableS
taskExecutor	null	Allows you to specify a custom task executor for consuming
taskExecutorSpring2	null	To use when using Spring 2.x with Camel. Allows you to spe messages.
templateConnectionFactory	null	The JMS connection factory used for sending messages.

`timeToLive`	`null`	When sending messages, specifies the time-to-live of the messa `explicitQosEnabled` option **must** also be enabled in order for
`transacted`	`false`	Specifies whether to use transacted mode for sending/receiving Pattern. See the section Enabling Transacted Consumption for
`transactedInOut`	`false`	**@deprecated:** Specifies whether to use transacted mode for send Pattern[14]. Applies only to producer endpoints. See section Enab details.
`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.
`transferException`	`false`	**Camel 2.0:** If enabled and you are using Request Reply messag the consumer side, then the caused `Exception` will be send bac `javax.jms.ObjectMessage`. If the client is Camel, the returned to use Camel JMS on page 293 as a bridge in your routing - for exa robust routing. Notice that if you also have **transferExchange** e The caught exception is required to be serializable. The original be wrapped in an outer exception such as `org.apache.camel.R` to the producer.
`transferExchange`	`false`	**Fuse Mediation Router 2.0:** You can transfer the exchange ove headers. The following fields are transferred: In body, Out body, Fault headers, exchange properties, exchange exception. This r Fuse Mediation Router will exclude any non-serializable objects
`username`	`null`	The username for the connector factory.
`useMessageIDAsCorrelationID`	`false`	Specifies whether `JMSMessageID` should always be used as `JMS`
`useVersion102`	`false`	**@deprecated (removed from Camel 2.5 onwards):** Specifies v

Message Mapping between JMS and Fuse Mediation Router

Fuse Mediation Router automatically maps messages between `javax.jms.Message` and `org.apache.camel.Message`.

When sending a JMS message, Fuse Mediation Router converts the message body to the following JMS message types:

Body Type	JMS Message	Comment
`String`	`javax.jms.TextMessage`	

[14] Exchange Pattern

`org.w3c.dom.Node`	`javax.jms.TextMessage`	The DOM will be converted to `String`.
`Map`	`javax.jms.MapMessage`	
`java.io.Serializable`	`javax.jms.ObjectMessage`	
`byte[]`	`javax.jms.BytesMessage`	
`java.io.File`	`javax.jms.BytesMessage`	
`java.io.Reader`	`javax.jms.BytesMessage`	
`java.io.InputStream`	`javax.jms.BytesMessage`	
`java.nio.ByteBuffer`	`javax.jms.BytesMessage`	

When receiving a JMS message, Fuse Mediation Router converts the JMS message to the following body type:

JMS Message	Body Type
`javax.jms.TextMessage`	`String`
`javax.jms.BytesMessage`	`byte[]`
`javax.jms.MapMessage`	`Map<String, Object>`
`javax.jms.ObjectMessage`	`Object`

Disabling auto-mapping of JMS messages

Available as of Fuse Mediation Router 1.6.2/2.0

You can use the `mapJmsMessage` option to disable the auto-mapping above. If disabled, Fuse Mediation Router will not try to map the received JMS message, but instead uses it directly as the payload. This allows you to avoid the overhead of mapping and let Fuse Mediation Router just pass through the JMS message. For instance, it even allows you to route `javax.jms.ObjectMessage` JMS messages with classes you do **not** have on the classpath.

Using a custom MessageConverter

Available as of Fuse Mediation Router 1.6.2/2.0

You can use the `messageConverter` option to do the mapping yourself in a Spring `org.springframework.jms.support.converter.MessageConverter` class.

For example, in the route below we use a custom message converter when sending a message to the JMS order queue:

```
  from("file://inbox/order").to("jms:queue:order?messageConverter=#myMessageConverter");
```

You can also use a custom message converter when consuming from a JMS destination.

Controlling the mapping strategy selected

Available as of Fuse Mediation Router 2.0

You can use the **jmsMessageType** option on the endpoint URL to force a specific message type for all messages. In the route below, we poll files from a folder and send them as `javax.jms.TextMessage` as we have forced the JMS producer endpoint to use text messages:

```
  from("file://inbox/order").to("jms:queue:order?jmsMessageType=Text");
```

You can also specify the message type to use for each messabe by setting the header with the key `CamelJmsMessageType`. For example:

```
  from("file://inbox/order").setHeader("CamelJmsMessageType", JmsMessage
Type.Text).to("jms:queue:order");
```

The possible values are defined in the `enum` class, `org.apache.camel.jms.JmsMessageType`.

Message format when sending

The exchange that is sent over the JMS wire must conform to the JMS Message spec[15].

For the `exchange.in.header` the following rules apply for the header **keys**:

- Keys starting with `JMS` or `JMSX` are reserved.
- `exchange.in.headers` keys must be literals and all be valid Java identifiers (do not use dots in the key name).
- From Fuse Mediation Router 1.4 until Fuse Mediation Router 1.6.x, Fuse Mediation Router automatically replaces all dots with underscores in key names. This replacement is reversed when Fuse Mediation Router consumes JMS messages.
- From Fuse Mediation Router 2.0 onwards, Fuse Mediation Router replaces dots & hyphens and the reverse when when consuming JMS messages: `.` is replaced by `_DOT_` and the reverse replacement when Fuse Mediation Router consumes the message. `-` is replaced by `_HYPHEN_` and the reverse replacement when Fuse Mediation Router consumes the message.
- See also the option `jmsKeyFormatStrategy` introduced in **Fuse Mediation Router 2.0**, which allows you to use your own custom strategy for formatting keys.

For the `exchange.in.header`, the following rules apply for the header **values**:

[15] http://java.sun.com/j2ee/1.4/docs/api/javax/jms/Message.html

- The values must be primitives or their counter objects (such as `Integer`, `Long`, `Character`). The types, `String`, `CharSequence`, `Date`, `BigDecimal` and `BigInteger` are all converted to their `toString()` representation. All other types are dropped.

Fuse Mediation Router will log with category `org.apache.camel.component.jms.JmsBinding` at **DEBUG** level if it drops a given header value. For example:

```
2008-07-09 06:43:04,046 [main           ] DEBUG JmsBinding
  - Ignoring non primitive header: order of class: org.apache.camel.component.jms.issues.Dummy
Order with value: DummyOrder{orderId=333, itemId=4444, quantity=2}
```

Message format when receiving

Fuse Mediation Router adds the following properties to the `Exchange` when it receives a message:

Property	Type	Description
`org.apache.camel.jms.replyDestination`	`javax.jms.Destination`	The reply destination.

Fuse Mediation Router adds the following JMS properties to the In message headers when it receives a JMS message:

Header	Type	Description
`JMSCorrelationID`	`String`	The JMS correlation ID.
`JMSDeliveryMode`	`int`	The JMS delivery mode.
`JMSDestination`	`javax.jms.Destination`	The JMS destination.
`JMSExpiration`	`long`	The JMS expiration.
`JMSMessageID`	`String`	The JMS unique message ID.
`JMSPriority`	`int`	The JMS priority (with 0 as the lowest priority and 9 as the highest).
`JMSRedelivered`	`boolean`	Is the JMS message redelivered.
`JMSReplyTo`	`javax.jms.Destination`	The JMS reply-to destination.
`JMSTimestamp`	`long`	The JMS timestamp.
`JMSType`	`String`	The JMS type.
`JMSXGroupID`	`String`	The JMS group ID.

As all the above information is standard JMS you can check the JMS documentation[16] for further details.

[16] http://java.sun.com/javaee/5/docs/api/javax/jms/Message.html

About using Fuse Mediation Router to send and receive messages and JMSReplyTo

The JMS component is complex and you have to pay close attention to how it works in some cases. So this is a short summary of some of the areas/pitfalls to look for.

When Fuse Mediation Router sends a message using its `JMSProducer`, it checks the following conditions:

- The message exchange pattern,
- Whether a `JMSReplyTo` was set in the endpoint or in the message headers,
- Whether any of the following options have been set on the JMS endpoint: `disableReplyTo`, `preserveMessageQos`, `explicitQosEnabled`.

All this can be a tad complex to understand and configure to support your use case.

JmsProducer

The `JmsProducer` behaves as follows, depending on configuration:

Exchange Pattern	Other options	Description
InOut		Fuse Mediation Router will expect a reply, set a temporary `JMSReplyTo`, and after sending the message, it will start to listen for the reply message on the temporary queue.
InOut	`JMSReplyTo` is set	Fuse Mediation Router will expect a reply and, after sending the message, it will start to listen for the reply message on the specified `JMSReplyTo` queue.
InOnly		Fuse Mediation Router will send the message and **not** expect a reply.
InOnly	`JMSReplyTo` is set	By default, Fuse Mediation Router suppresses the `JMSReplyTo` destination and clears the `JMSReplyTo` header before sending the message. Fuse Mediation Router then sends the message and does not expect a reply. Fuse Mediation Router logs this in the log at `DEBUG` level and you should see: `DEBUG JmsProducer - Disabling JMSReplyTo as this Exchange is not OUT capable with JMSReplyTo: myReplyQueue to destination: myQueue`. If you want to leave the `JMSReplyTo` header in the outgoing message, you must set either `preserveMessageQos=true` or `explicitQosEnabled=true`. From Fuse Mediation Router 2.6 onwards, you can also populate the `JMSReplyTo` header by setting the `replyTo` option in the URI. For example, if you send a message to the

	`jms:queue:Foo?replyTo=FooReply&preserveMessageQos=true` URI, the `JMSReplyTo` header is included, even if the exchange is *InOnly*.

JmsConsumer

The `JmsConsumer` behaves as follows, depending on configuration:

Exchange Pattern	Other options	Description
InOut		Fuse Mediation Router will send the reply back to the `JMSReplyTo` queue.
InOnly		Fuse Mediation Router will not send a reply back, as the pattern is *InOnly*.
	`disableReplyTo=true`	This option suppresses replies.

So pay attention to the message exchange pattern set on your exchanges.

If you send a message to a JMS destination in the middle of your route you can specify the exchange pattern to use, see more at Request Reply. This is useful if you want to send an `InOnly` message to a JMS topic:

```
from("activemq:queue:in")
   .to("bean:validateOrder")
   .to(ExchangePattern.InOnly, "activemq:topic:order")
   .to("bean:handleOrder");
```

Reuse endpoint and send to different destinations computed at runtime

Available as of Fuse Mediation Router 1.6.2/2.0 If you need to send messages to a lot of different JMS destinations, it makes sense to reuse a JMS endpoint and specify the real destination in a message header. This allows Fuse Mediation Router to reuse the same endpoint, but send to different destinations. This greatly reduces the number of endpoints created and economizes on memory and thread resources.

You can specify the destination in the following headers:

Header	Type	Description
`CamelJmsDestination`	`javax.jms.Destination`	**Fuse Mediation Router 2.0:** A destination object.
`CamelJmsDestinationName`	`String`	**Fuse Mediation Router 1.6.2/2.0:** The destination name.

For example, the following route shows how you can compute a destination at run time and use it to override the destination appearing in the JMS URL:

```
from("file://inbox")
  .to("bean:computeDestination")
  .to("activemq:queue:dummy");
```

The queue name, `dummy`, is just a placeholder. It must be provided as part of the JMS endpoint URL, but it will be ignored in this example.

In the `computeDestination` bean, specify the real destination by setting the `CamelJmsDestinationName` header as follows:

```
public void setJmsHeader(Exchange exchange) {
   String id = ....
   exchange.getIn().setHeader("CamelJmsDestinationName", "order:" + id");
}
```

Then Fuse Mediation Router will read this header and use it as the destination instead of the one configured on the endpoint. So, in this example Fuse Mediation Router sends the message to `activemq:queue:order:2`, assuming the `id` value was 2.

If both the `CamelJmsDestination` and the `CamelJmsDestinationName` headers are set, `CamelJmsDestination` takes priority.

Configuring different JMS providers

You can configure your JMS provider in Spring XML as follows:

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

<bean id="activemq" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://localhost?broker.persistent=false"/>
    </bean>
  </property>
</bean>
```

Basically, you can configure as many JMS component instances as you wish and give them a unique name using the **id attribute**. The preceding example configures an `activemq` component. You could do the same to configure MQSeries, TibCo, BEA, Sonic and so on.

Once you have a named JMS component, you can then refer to endpoints within that component using URIs. For example for the component name, `activemq`, you can then refer to destinations using the URI format, `activemq:[queue:|topic:]destinationName`. You can use the same approach for all other JMS providers.

This works by the SpringCamelContext lazily fetching components from the spring context for the scheme name you use for Endpoint URIs and having the Component resolve the endpoint URIs.

Using JNDI to find the ConnectionFactory

If you are using a J2EE container, you might need to look up JNDI to find the JMS `ConnectionFactory` rather than use the usual `<bean>` mechanism in Spring. You can do this using Spring's factory bean or the new Spring XML namespace. For example:

```
<bean id="weblogic" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory" ref="myConnectionFactory"/>
</bean>

<jee:jndi-lookup id="myConnectionFactory" jndi-name="jms/connectionFactory"/>
```

See The jee schema[17] in the Spring reference documentation for more details about JNDI lookup.

Using JNDI to lookup the physical queues

You need to use the `destinationResolver` option to use the Spring JNDI resolver that can lookup in the JNDI, or use your own custom implementation.

See this nabble post for more details:
http://www.nabble.com/JMS-queue---JNDI-instead-of-physical-name-td24484994.html [18]

Using WebSphere MQ

See this link at nabble[19] for details of how a Fuse Mediation Router user configured JMS on page 293 to connect to remote WebSphere MQ brokers.

Concurrent Consuming

A common requirement with JMS is to consume messages concurrently in multiple threads in order to make an application more responsive. You can set the `concurrentConsumers` option to specify the number of threads servicing the JMS endpoint, as follows:

```
from("jms:SomeQueue?concurrentConsumers=20").
  bean(MyClass.class);
```

You can configure this option in one of the following ways:

- On the `JmsComponent`,
- On the endpoint URI or,

[17] http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/apcs02.html#xsd-config-body-schemas-jee
[18] http://www.nabble.com/JMS-queue---JNDI-instead-of-physical-name-td24484994.html
[19] http://www.nabble.com/Camel-and-IBM-MQ-Series-td24524277.html

- By invoking `setConcurrentConsumers()` directly on the `JmsEndpoint`.

Enabling Transacted Consumption

A common requirement is to consume from a queue in a transaction and then process the message using the Fuse Mediation Router route. To do this, just ensure that you set the following properties on the component/endpoint:

- `transacted` = true
- `transactionManager` = a Transsaction Manager - typically the `JmsTransactionManager`

See also the Transactional Client EIP pattern for further details.

Using JMSReplyTo for late replies

Avaiable as of Fuse Mediation Router 2.0

When using Fuse Mediation Router as a JMS listener, it sets an Exchange property with the value of the ReplyTo `javax.jms.Destination` object, having the key `ReplyTo`. You can obtain this `Destination` as follows:

```
Destination replyDestination = exchange.getIn().getHeader(JmsConstants.JMS_REPLY_DESTINATION,
 Destination.class);
```

And then later use it to send a reply using regular JMS or Fuse Mediation Router.

```
    // we need to pass in the JMS component, and in this sample we use ActiveMQ
    JmsEndpoint endpoint = JmsEndpoint.newInstance(replyDestination, activeMQComponent);
    // now we have the endpoint we can use regular Fuse Mediation Router API to send a
message to it
    template.sendBody(endpoint, "Here is the late reply.");
```

A different solution to sending a reply is to provide the `replyDestination` object in the same Exchange property when sending. Fuse Mediation Router will then pick up this property and use it for the real destination. The endpoint URI must include a dummy destination, however. For example:

```
    // we pretend to send it to some non existing dummy queue
    template.send("activemq:queue:dummy, new Processor() {
        public void process(Exchange exchange) throws Exception {
            // and here we override the destination with the ReplyTo destination object so
 the message is sent to there instead of dummy
            exchange.getIn().setHeader(JmsConstants.JMS_DESTINATION, replyDestination);
            exchange.getIn().setBody("Here is the late reply.");
        }
    }
```

Using a request timeout

In the sample below we send a Request Reply style message Exchange (we use the `requestBody` method = `InOut`) to the slow queue for further processing in Camel and we wait for a return reply:

```
// send a in-out with a timeout for 5 sec
Object out = template.requestBody("activemq:queue:slow?requestTimeout=5000", "Hello World");
```

Samples

JMS is used in many examples for other components as well. But we provide a few samples below to get started.

Receiving from JMS

In the following sample we configure a route that receives JMS messages and routes the message to a POJO:

```
   from("jms:queue:foo").
     to("bean:myBusinessLogic");
```

You can of course use any of the EIP patterns so the route can be context based. For example, here's how to filter an order topic for the big spenders:

```
from("jms:topic:OrdersTopic").
  filter().method("myBean", "isGoldCustomer").
    to("jms:queue:BigSpendersQueue");
```

Sending to a JMS

In the sample below we poll a file folder and send the file content to a JMS topic. As we want the content of the file as a `TextMessage` instead of a `BytesMessage`, we need to convert the body to a `String`:

```
from("file://orders").
  convertBodyTo(String.class).
  to("jms:topic:OrdersTopic");
```

Using Annotations

Fuse Mediation Router also has annotations so you can use POJO Consuming[21] and POJO Producing.

[21] POJO Consuming

Spring DSL sample

The preceding examples use the Java DSL. Fuse Mediation Router also supports Spring XML DSL. Here is the big spender sample using Spring DSL:

```
<route>
  <from uri="jms:topic:OrdersTopic"/>
  <filter>
    <method bean="myBean" method="isGoldCustomer"/>
    <to uri="jms:queue:BigSpendersQueue"/>
  </filter>
</route>
```

Other samples

JMS appears in many of the examples for other components and EIP patterns, as well in this Fuse Mediation Router documentation. So feel free to browse the documentation. If you have time, check out the this tutorial that uses JMS but focuses on how well Spring Remoting and Fuse Mediation Router works together Tutorial-JmsRemoting.

Using JMS as a Dead Letter Queue storing Exchange

Available as of Camel 2.0 Normally, when using JMS on page 293 as the transport, it only transfers the body and headers as the payload. If you want to use JMS on page 293 with a Dead Letter Channel, using a JMS queue as the Dead Letter Queue, then normally the caused Exception is not stored in the JMS message. You can, however, use the **transferExchange** option on the JMS dead letter queue to instruct Camel to store the entire Exchange in the queue as a `javax.jms.ObjectMessage` that holds a `org.apache.camel.impl.DefaultExchangeHolder`. This allows you to consume from the Dead Letter Queue and retrieve the caused exception from the Exchange property with the key `Exchange.EXCEPTION_CAUGHT`. The demo below illustrates this:

```
// setup error handler to use JMS as queue and store the entire Exchange
errorHandler(deadLetterChannel("jms:queue:dead?transferExchange=true"));
```

Then you can consume from the JMS queue and analyze the problem:

```
from("jms:queue:dead").to("bean:myErrorAnalyzer");

// and in our bean
String body = exchange.getIn().getBody();
Exception cause = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.class);
// the cause message is
String problem = cause.getMessage();
```

Using JMS as a Dead Letter Channel storing error only

You can use JMS to store the cause error message or to store a custom body, which you can initialize yourself. The following example uses the Message Translator EIP to do a transformation on the failed exchange before it is moved to the JMS on page 293 dead letter queue:

```
// we sent it to a seda dead queue first
errorHandler(deadLetterChannel("seda:dead"));

// and on the seda dead queue we can do the custom transformation before its sent to the
JMS queue
from("seda:dead").transform(exceptionMessage()).to("jms:queue:dead");
```

Here we only store the original cause error message in the transform. You can, however, use any Expression to send whatever you like. For example, you can invoke a method on a Bean or use a custom processor.

- Transactional Client
- Bean Integration
- Tutorial-JmsRemoting
- JMSTemplate gotchas[22]

Sending an InOnly message and keeping the JMSReplyTo header

When sending to a JMS on page 293 destination using **camel-jms** the producer will use the MEP to detect if its InOnly or InOut messaging. However there can be times where you want to send an InOnly message but keeping the JMSReplyTo header. To do so you have to instruct Camel to keep it, otherwise the JMSReplyTo header will be dropped.

For example to send an InOnly message to the foo queue, but with a JMSReplyTo with bar queue you can do as follows:

```
    template.send("activemq:queue:foo?preserveMessageQos=true", new Processor() {
    public void process(Exchange exchange) throws Exception {
    exchange.getIn().setBody("World");
    exchange.getIn().setHeader("JMSReplyTo", "bar");
    }
    });
```

[22] http://activemq.apache.org/jmstemplate-gotchas.html

Notice we use `preserveMessageQos=true` to instruct Camel to keep the JMSReplyTo header.

Chapter 46. JMX

JMX Component

The JMX component enables consumers to subscribe to an MBean's notifications. The component supports passing the `Notification` object directly through the exchange or serializing it to XML according to the schema provided within this project. This is a consumer-only component. Exceptions are thrown if you attempt to create a producer for it.

URI Format

The component can connect to the local platform MBean server with the following URI:

```
jmx://platform?options
```

A remote MBean server URL can be specified after the `jmx:` scheme prefix, as follows:

```
jmx:service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi?options
```

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

URI Options

Property	Required	Default	Description
`format`		`xml`	Format for the message body. Either `xml` or `raw`. If `xml`, the notification is serialized to XML. If `raw`, the raw java object is set as the body.
`password`			Credentials for making a remote connection.
`objectDomain`	*Yes*		The domain of the MBean you are connecting to.
`objectName`			The name key for the MBean you are connecting to. Either this property of a list of keys must be provided (but not both). For more details, see "ObjectName Construction" on page 314.
`notificationFilter`			Reference to a bean that implements the `NotificationFilter` interface. The `#beanID` syntax should be used to reference the bean in the registry.
`handback`			Value to hand back to the listener when a notification is received. This value will be put into the `jmx.handback` message header.

ObjectName Construction

The URI must always have the `objectDomain` property. In addition, the URI must contain either `objectName` or one or more properties that start with `key`.

Domain with Name property

When the `objectName` property is provided, the following constructor is used to build the `ObjectName` instance for the MBean:

```
ObjectName(String domain, String key, String value)
```

The `key` value in the preceding constructor must be `name` and the value is the value of the `objectName` property.

Domain with Hashtable

```
ObjectName(String domain, Hashtable<String,String> table)
```

The `Hashtable` is constructed by extracting properties that start with `key`. The properties will have the `key` prefix stripped prior to building the `Hashtable`. This allows the URI to contain a variable number of properties to identify the MBean.

Example

```
from("jmx:platform?objectDomain=jmxExample&key.name=simpleBean").
        to("log:jmxEvent");
```

Full example

A complete example using the JMX component is available under the `examples/camel-example-jmx` directory.

Chapter 47. JPA

JPA Component

The **jpa** component enables you to store and retrieve Java objects from persistent storage using EJB 3's Java Persistence Architecture (JPA), which is a standard interface layer that wraps Object/Relational Mapping (ORM) products such as OpenJPA, Hibernate, TopLink, and so on.

Sending to the endpoint

You can store a Java entity bean in a database by sending it to a JPA producer endpoint. The body of the *In* message is assumed to be an entity bean (that is, a POJO with an @Entity[1] annotation on it) or a collection or an array of entity beans.

If the body does not contain one of the preceding types, put a Message TranslatorMessage Translator in front of the endpoint to perform the necessary conversion first.

Consuming from the endpoint

Consuming messages from a JPA consumer endpoint removes (or updates) entity beans in the database. This allows you to use a database table as a logical queue: consumers take messages from the queue and then delete/update them to logically remove them from the queue.

If you do not wish to delete the entity bean when it has been processed, you can specify `consumeDelete=false` on the URI. This will result in the entity being processed each poll.

If you would rather perform some update on the entity to mark it as processed (such as to exclude it from a future query) then you can annotate a method with @Consumed[2] which will be invoked on your entity bean when the entity bean is consumed.

URI format

```
jpa:[entityClassName][?options]
```

For sending to the endpoint, the *entityClassName* is optional. If specified, it helps the Type Converter to ensure the body is of the correct type.

For consuming, the *entityClassName* is mandatory.

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

[1] http://java.sun.com/javaee/5/docs/api/javax/persistence/Entity.html
[2] http://camel.apache.org/maven/current/camel-jpa/apidocs/org/apache/camel/component/jpa/Consumed.html

Options

Name	Default Value	Description
`entityType`	*entityClassName*	Overrides the *entityClassName* from the URI.
`persistenceUnit`	`camel`	The JPA persistence unit used by default.
`consumeDelete`	`true`	**JPA consumer only:** If `true`, the entity is deleted after it is consumed; if `false`, the entity is not deleted.
`consumeLockEntity`	`true`	**JPA consumer only:** Specifies whether or not to set an exclusive lock on each entity bean while processing the results from polling.
`flushOnSend`	`true`	**JPA producer only:** Flushes the EntityManager[3] after the entity bean has been persisted.
`maximumResults`	`-1`	**JPA consumer only:** Set the maximum number of results to retrieve on the Query[4].
`transactionManager`	`null`	**Fuse Mediation Router 1.6.1/2.0:** Specifies the transaction manager to use. If none provided, Fuse Mediation Router will use a `JpaTransactionManager` by default. Can be used to set a JTA transaction manager (for integration with an EJB container).
`consumer.delay`	`500`	**JPA consumer only:** Delay in milliseconds between each poll.
`consumer.initialDelay`	`1000`	**JPA consumer only:** Milliseconds before polling starts.
`consumer.useFixedDelay`	`false`	**JPA consumer only:** Set to `true` to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService[5] in JDK for details.
`maxMessagesPerPoll`	`0`	**Fuse Mediation Router 2.0:JPA consumer only:** An integer value to define the maximum number of messages to gather per poll. By default, no maximum is set. Can be used to avoid polling many thousands of messages when starting up the server. Set a value of 0 or negative to disable.
`consumer.query`		**JPA consumer only:** To use a custom query when consuming data.
`consumer.namedQuery`		**JPA consumer only:** To use a named query when consuming data.

[3] http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html
[4] http://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html
[5] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html

`consumer.nativeQuery`		**JPA consumer only:** To use a custom native query when consuming data.
`usePersist`	`false`	**Camel 2.5: JPA producer only:** Indicates to use `entityManager.persist(entity)` instead of `entityManager.merge(entity)`. Note: `entityManager.persist(entity)` doesn't work for detached entities (where the EntityManager has to execute an UPDATE instead of an INSERT query)!

Message Headers

Fuse Mediation Router adds the following message headers to the exchange:

Header	Type	Description
`CamelJpaTemplate`	`JpaTemplate`	**Fuse Mediation Router 2.0:** The `JpaTemplate` object that is used to access the entity bean. You need this object in some situations, for instance in a type converter or when you are doing some custom processing.

Configuring EntityManagerFactory

You are strongly advised to configure the JPA component to use a specific `EntityManagerFactory` instance. If you do not do so, each `JpaEndpoint` will auto-create its own `EntityManagerFactory` instance.For example, you can instantiate a JPA component that references the `myEMFactory` entity manager factory, as follows:

```
<bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent">
   <property name="entityManagerFactory" ref="myEMFactory"/>
</bean>
```

In **Camel 2.3** the `JpaComponent` will auto lookup the `EntityManagerFactory` from the Registry which means you do not need to configure this on the `JpaComponent` as shown above. You only need to do so if there is ambiguity, in which case Camel will log a WARN.

Configuring TransactionManager

You are strongly advised to specify the `TransactionManager` instance used by the JPA component. If you do not do so, each `JpaEndpoint` will auto-create its own instance of `TransactionManager`. For example, you can instantiate a JPA component that references the `myTransactionManager` transaction manager, as follows:

```
<bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent">
   <property name="entityManagerFactory" ref="myEMFactory"/>
```

```
   <property name="transactionManager" ref="myTransactionManager"/>
</bean>
```

In **Camel 2.3** the `JpaComponent` will auto lookup the `TransactionManager` from the Registry which means you do not need to configure this on the `JpaComponent` as shown above. You only need to do so if there is ambiguity, in which case Camel will log a WARN.

Using a consumer with a named query

For consuming only selected entities, you can use the `consumer.namedQuery` URI query option. First, you have to define the named query in the JPA Entity class:

```
@Entity
@NamedQuery(name = "step1", query = "select x from MultiSteps x where x.step = 1")
public class MultiSteps {
   ...
}
```

After that you can define a consumer uri like this one:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.namedQuery=step1")
.to("bean:myBusinessLogic");
```

Using a consumer with a query

For consuming only selected entities, you can use the `consumer.query` URI query option. You only have to define the query option:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.query=select o from
org.apache.camel.examples.MultiSteps o where o.step = 1")
.to("bean:myBusinessLogic");
```

Using a consumer with a native query

For consuming only selected entities, you can use the `consumer.nativeQuery` URI query option. You only have to define the native query option:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.nativeQuery=select * from MultiSteps
 where step = 1")
```

```
.to("bean:myBusinessLogic");
```

If you use the native query option, you will receive an object array in the message body.

Example

See the Tracer Example for an example using JPA to store traced messages into a database.

Chapter 48. JT400

JT/400 Component

The **`jt400`** component allows you to exchanges messages with an AS/400 system using data queues. This components is only available in Fuse Mediation Router 1.5 and above.

URI format

```
jt400://user:password@system/QSYS.LIB/LIBRARY.LIB/QUEUE.DTAQ[?options]
```

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

URI options

Name	Default value	Description
`ccsid`	default system CCSID	Specifies the CCSID to use for the connection with the AS/400 system.
`format`	`text`	Specifies the data format for sending messages valid options are: `text` (represented by `String`) and `binary` (represented by `byte[]`)
`consumer.delay`	`500`	Delay in milliseconds between each poll.
`consumer.initialDelay`	`1000`	Milliseconds before polling starts.
`consumer.userFixedDelay`	`false`	`true` to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService[1] in JDK for details.

Usage

When configured as a consumer endpoint, the endpoint will poll a data queue on a remote system. For every entry on the data queue, a new `Exchange` is sent with the entry's data in the *In* message's body, formatted either as a `String` or a `byte[]`, depending on the format. For a provider endpoint, the *In* message body contents will be put on the data queue as either raw bytes or text.

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html

Example

In the snippet below, the data for an exchange sent to the `direct:george` endpoint will be put in the data queue `PENNYLANE` in library `BEATLES` on a system named `LIVERPOOL`. Another user connects to the same data queue to receive the information from the data queue and forward it to the `mock:ringo` endpoint.

```
public class Jt400RouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
       from("direct:george").to("jt400://GEORGE:EGROEG@LIVER
POOL/QSYS.LIB/BEATLES.LIB/PENNYLANE.DTAQ");
       from("jt400://RINGO:OGNIR@LIVER
POOL/QSYS.LIB/BEATLES.LIB/PENNYLANE.DTAQ").to("mock:ringo");
    }
}
```

Chapter 49. Language

Language

Available as of Camel 2.5

The language component allows you to send Exchange to an endpoint which executes a script by any of the supported Languages in Camel. By having a component to execute language scripts, it allows more dynamic routing capabilities. For example by using the Routing SlipRouting Slip or Dynamic RouterDynamic Router EIPs you can send messages to `language` endpoints where the script is dynamic defined as well.

This component is provided out of the box in `camel-core` and hence no additional JARs is needed. You only have to include additional Camel components if the language of choice mandates it, such as using Groovy or JavaScript languages.

URI format

```
language://languageName[:script][?options]
```

URI Options

The component supports the following options.

Name	Default Value	Type	Description
`languageName`	`null`	`String`	The name of the Language[1] to use, such as `simple`, `groovy`, `javascript` etc. This option is mandatory.
`script`	`null`	`String`	The script to execute.
`transform`	`true`	`boolean`	Whether or not the result of the script should be used as the new message body. By setting to `false` the script is executed but the result of the script is discarded.

Message Headers

The following message headers can be used to affect the behavior of the component

[1] Languages

Header	Description
`CamelLanguageScript`	The script to execute provided in the header. Takes precedence over script configured on the endpoint.

Examples

For example you can use the Simple language to Message TranslatorMessage Translator a message:

```
from("direct:start").to("language:simple:Hello ${body}").to("mock:result");
```

In case you want to convert the message body type you can do this as well:

```
from("direct:start").to("language:simple:${mandatoryBodyAs(String)}").to("mock:result");
```

You can also use the Groovy language, such as this example where the input message will by multiplied with 2:

```
from("direct:start").to("language:groovy:request.body * 2").to("mock:result");
```

You can also provide the script as a header as shown below. Here we use XPath language to extract the text from the `<foo>` tag.

```
Object out = producer.requestBodyAndHeader("language:xpath", "<foo>Hello World</foo>", Ex
change.LANGUAGE_SCRIPT, "/foo/text()");
assertEquals("Hello World", out);
```

- Languages
- Routing SlipRouting Slip
- Dynamic RouterDynamic Router

Chapter 50. LDAP

LDAP Component

The **ldap** component allows you to perform searches in LDAP servers using filters as the message payload. This component uses standard JNDI (`javax.naming` package) to access the server.

URI format

```
ldap:ldapServerBean[?options]
```

The *ldapServerBean* portion of the URI refers to a DirContext[1] bean in the registry. The LDAP component only supports producer endpoints, which means that an `ldap` URI cannot appear in the `from` at the start of a route.

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

Options

Name	Default Value	Description
`base`	`ou=system`	The base DN for searches.
`scope`	`subtree`	Specifies how deeply to search the tree of entries, starting at the base DN. Value can be `object`, `onelevel`, or `subtree`.
`pageSize`	No paging used.	When specified the LDAP module uses paging to retrieve all results (most LDAP Servers throw an exception when trying to retrieve more than 1000 entries in one query). To be able to use this, an `LdapContext` (subclass of `DirContext`) has to be passed in as `ldapServerBean` (otherwise an exception is thrown)
`returnedAttributes`	Depends on LDAP Server (could be all or none) .	Comma-separated list of attributes that should be set in each entry of the result

Result

The result is returned in the Out body as a `ArrayList<javax.naming.directory.SearchResult>` object.

[1] http://java.sun.com/j2se/1.4.2/docs/api/javax/naming/directory/DirContext.html

DirContext

The URI, `ldap:ldapserver`, references a Spring bean with the ID, `ldapserver`. The `ldapserver` bean may be defined as follows:

```
<bean id="ldapserver" class="javax.naming.directory.InitialDirContext" scope="prototype">
  <constructor-arg>
    <props>
      <prop key="java.naming.factory.initial">com.sun.jndi.ldap.LdapCtxFactory</prop>
      <prop key="java.naming.provider.url">ldap://localhost:10389</prop>
      <prop key="java.naming.security.authentication">none</prop>
    </props>
  </constructor-arg>
</bean>
```

The preceding example declares a regular Sun based LDAP `DirContext` that connects anonymously to a locally hosted LDAP server.

Note

`DirContext` objects are **not** required to support concurrency by contract. It is therefore important that the directory context is declared with the setting, `scope="prototype"`, in the `bean` definition or that the context supports concurrency. In the Spring framework, `prototype` scoped objects are instantiated each time they are looked up.

Note

Fuse Mediation Router 1.6.1 and Fuse Mediation Router 2.0 include a fix[2] to support concurrency for LDAP producers. *ldapServerBean* contexts are now looked up each time a request is sent to the LDAP server. In addition, the contexts are released as soon as the producer completes.

Samples

Following on from the Spring configuration above, the code sample below sends an LDAP request to filter search a group for a member. The Common Name is then extracted from the response.

[2] https://issues.apache.org/activemq/browse/CAMEL-1583?page=com.atlassian.jira.plugin.system.issuetabpanels:comment-tabpanel&focusedCommentId=51503

```
ProducerTemplate<Exchange> template = exchange
  .getContext().createProducerTemplate();

Collection<?> results = (Collection<?>) (template
  .sendBody(
    "ldap:ldapserver?base=ou=mygroup,ou=groups,ou=system",
    "(member=uid=huntc,ou=users,ou=system)"));

if (results.size() > 0) {
  // Extract what we need from the device's profile

  Iterator<?> resultIter = results.iterator();
  SearchResult searchResult = (SearchResult) resultIter
      .next();
  Attributes attributes = searchResult
      .getAttributes();
  Attribute deviceCNAttr = attributes.get("cn");
  String deviceCN = (String) deviceCNAttr.get();

  ...
```

If no specific filter is required - for example, you just need to look up a single entry - specify a wildcard filter expression. For example, if the LDAP entry has a Common Name, use a filter expression like:

```
(cn=*)
```

Binding using credentials

A Camel end user donated this sample code he used to bind to the ldap server using credentials.

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
props.setProperty(Context.PROVIDER_URL, "ldap://localhost:389");
props.setProperty(Context.URL_PKG_PREFIXES, "com.sun.jndi.url");
props.setProperty(Context.REFERRAL, "ignore");
props.setProperty(Context.SECURITY_AUTHENTICATION, "simple");
props.setProperty(Context.SECURITY_PRINCIPAL, "cn=Manager");
props.setProperty(Context.SECURITY_CREDENTIALS, "secret");

SimpleRegistry reg = new SimpleRegistry();
reg.put("myldap", new InitialLdapContext(props, null));

CamelContext context = new DefaultCamelContext(reg);
context.addRoutes(
    new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:start").to("ldap:myldap?base=ou=test");
        }
```

```
    }
);
context.start();

ProducerTemplate template = context.createProducerTemplate();

Endpoint endpoint = context.getEndpoint("direct:start");
Exchange exchange = endpoint.createExchange();
exchange.getIn().setBody("(uid=test)");
Exchange out = template.send(endpoint, exchange);

Collection<SearchResult> data = out.getOut().getBody(Collection.class);
assert data != null;
assert !data.isEmpty();

System.out.println(out.getOut().getBody());

context.stop();
```

Chapter 51. List

List Component

deprecated: is renamed to the Browse on page 51 component in Fuse Mediation Router 2.0

The List component provides a simple BrowsableEndpoint which can be useful for testing, visualisation tools or debugging. The exchanges sent to the endpoint are all available to be browsed.

URI format

```
list:someName
```

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

Sample

In the route below we have the list component to be able to browse the Exchanges that is passed through:

```
  from("activemq:order.in").to("list:orderReceived").to("bean:processOrder");
```

Then we will be able to inspect the received exchanges from java code:

```
    private CamelContext context;

    public void inspectRecievedOrders() {
        BrowsableEndpoint browse = context.getEndpoint("list:orderReceived", BrowsableEnd
point.class);
        List<Exchange> exchanges = browse.getExchanges();
        ...
        // then we can inspect the list of received exchanges from Java
        for (Exchange exchange : exchanges) {
            String payload = exchange.getIn().getBody();
            ...
        }
   }
```

See also:

- Browse on page 51

Chapter 52. Log

Log Component

The **log:** component logs message exchanges to the underlying logging mechanism.

URI format

```
log:loggingCategory[?options]
```

Where **loggingCategory** is the name of the logging category to use. You can append query options to the URI in the following format, `?option=value&option=value&...`

For example, a log endpoint typically specifies the logging level using the `level` option, as follows:

```
log:org.apache.camel.example?level=DEBUG
```

The default logger logs every exchange (*regular logging*). But Fuse Mediation Router also ships with the `Throughput` logger, which is used whenever the `groupSize` option is specified.

Also a log in the DSL

In **Camel 2.2** onwards there is a `log` directly in the DSL, but it has a different purpose. Its meant for lightweight and human logs. See more details at LogEIP.

Options

Option	Default	Type	Description
`level`	`INFO`	`String`	Logging level to use. Possible values: `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`, `OFF`
`groupSize`	`null`	`Integer`	An integer that specifies a group size for throughput logging.
`groupInterval`	`null`	`Integer`	**Camel 2.6**: If specified will group message stats by this time interval (in millis)
`groupDelay`	`0`	`Integer`	**Camel 2.6**: Set the initial delay for stats (in millis)
`groupActiveOnly`	`true`	`boolean`	**Camel 2.6**: If true, will hide stats when no new messages have been received for a time interval, if false, show stats regardless of message traffic

Note

`groupDelay` and `groupActiveOnly` are only applicable when using `groupInterval`.

Formatting

The log formats the execution of exchanges to log lines. By default, the log uses `LogFormatter` to format the log output, where `LogFormatter` has the following options:

Option	Default	Description
`showAll`	`false`	Quick option for turning all options on (multiline, maxChars has to be manually set if to be used).
`showExchangeId`	`false`	Show the unique exchange ID.
`showExchangePattern`	`true`	**Camel 2.3:** Shows the Message Exchange Pattern (or MEP for short).
`showProperties`	`false`	Show the exchange properties.
`showHeaders`	`false`	Show the In message headers.
`showBodyType`	`true`	Show the In body Java type.
`showBody`	`true`	Show the In body.
`showOut`	`false`	If the exchange has an Out message, show the Out message.
`showException`	`false`	**Fuse Mediation Router 2.0:** If the exchange has an exception, show the exception message (no stack trace).
`showCaughtException`	`false`	**Fuse Mediation Router 2.0:** If the exchange has a caught exception, show the exception message (no stack trace). A caught exception is stored as a property on the exchange and for instance a `doCatch` can catch exceptions. See Try Catch Finally.
`showStackTrace`	`false`	**Fuse Mediation Router 2.0:** Show the stack trace, if an exchange has an exception. Only effective if one of `showAll`, `showException` or `showCaughtException` are enabled.
`showFuture`	`false`	**Camel 2.1:** Whether Camel should show `java.util.concurrent.Future` bodies or not. If enabled Camel could potentially wait until the `Future` task is done. Will by default not wait.
`multiline`	`false`	If `true`, each piece of information is logged on a new line.
`maxChars`		**Fuse Mediation Router 2.0:** Limits the number of characters logged per line.

Regular logger sample

In the route below we log the incoming orders at `DEBUG` level before the order is processed:

```
from("activemq:orders").to("log:com.mycompany.order?level=DEBUG").to("bean:processOrder");
```

Or using Spring XML to define the route:

```
  <route>
    <from uri="activemq:orders"/>
    <to uri="log:com.mycompany.order?level=DEBUG"/>
    <to uri="bean:processOrder"/>
  </route>
```

Regular logger with formatter sample

In the route below we log the incoming orders at `INFO` level before the order is processed.

```
from("activemq:orders").
    to("log:com.mycompany.order?showAll=true&multiline=true").to("bean:processOrder");
```

Throughput logger with groupSize sample

In the route below we log the throughput of the incoming orders at `DEBUG` level grouped by 10 messages.

```
from("activemq:orders").
    to("log:com.mycompany.order?level=DEBUG?groupSize=10").to("bean:processOrder");
```

Throughput logger with groupInterval sample

This route will result in message stats logged every 10s, with an initial 60s delay and stats should be displayed even if there isn't any message traffic.

```
from("activemq:orders")
    .to("log:com.mycompany.order?level=DEBUG?groupInterval=10000&groupDelay=60000&groupAct
iveOnly=false")
    .to("bean:processOrder");
```

The following will be logged:

```
"Received: 1000 new messages, with total 2000 so far. Last group took: 10000 millis which
is: 100 messages per second. average: 100"
```

Chapter 53. Lucene

Lucene (Indexer and Search) Component

Available as of Fuse Mediation Router 2.2

The **lucene** component is based on the Apache Lucene project. Apache Lucene is a powerful high-performance, full-featured text search engine library written entirely in Java. For more details about Lucene, please see the following links * http://lucene.apache.org/java/docs/ [1] * http://lucene.apache.org/java/docs/features.html [2]

The lucene component in camel facilitates integration and utilization of Lucene endpoints in enterprise integration patterns and scenarios. The lucene component does the following

- builds a searchable index of documents when payloads are sent to the Lucene Endpoint
- facilitates performing of indexed searches in Fuse Mediation Router

This component only supports producer endpoints.

URI format

```
lucene:searcherName:insert[?options]
lucene:searcherName:query[?options]
```

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

Insert Options

Name	Default Value	Description
`analyzer`	`StandardAnalyzer`	An Analyzer builds TokenStreams, which analyze text. It thus represents a policy for extracting index terms from text. The value for analyzer can be any class that extends the abstract class org.apache.lucene.analysis.Analyzer. Lucene also offers a rich set of analyzers out of the box
`indexDir`	`./indexDirectory`	A file system directory in which index files are created upon analysis of the document by the specified analyzer
`srcDir`	`null`	An optional directory containing files to be used to be analyzed and added to the index at producer startup.

[1] http://lucene.apache.org/java/docs/
[2] http://lucene.apache.org/java/docs/features.html

Query Options

Name	Default Value	Description
`analyzer`	`StandardAnalyzer`	An Analyzer builds TokenStreams, which analyze text. It thus represents a policy for extracting index terms from text. The value for analyzer can be any class that extends the abstract class org.apache.lucene.analysis.Analyzer. Lucene also offers a rich set of analyzers out of the box
`indexDir`	`./indexDirectory`	A file system directory in which index files are created upon analysis of the document by the specified analyzer
`maxHits`	`10`	An integer value that limits the result set of the search operation

Message Headers

Header	Description
`QUERY`	The Lucene Query to performed on the index. The query may include wildcards and phrases

Lucene Producers

This component supports 2 producer endpoints.

- **insert** - The insert producer builds a searchable index by analyzing the body in incoming exchanges and associating it with a token ("content").
- **query** - The query producer performs searches on a pre-created index. The query uses the searchable index to perform score & relevance based searches. Queries are sent via the incoming exchange contains a header property name called 'QUERY'. The value of the header property 'QUERY' is a Lucene Query. For more details on how to create Lucene Queries check out http://lucene.apache.org/java/3_0_0/queryparsersyntax.html [3]

Lucene Processor

There is a processor called LuceneQueryProcessor available to perform queries against lucene without the need to create a producer.

[3] http://lucene.apache.org/java/3_0_0/queryparsersyntax.html

Example 1: Creating a Lucene index

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
       from("direct:start").
           to("lucene:whitespaceQuotesIndex:insert?analyzer=#whitespaceAnalyzer&in
dexDir=#whitespace&srcDir=#load_dir").
           to("mock:result");
    }
};
```

Example 2: Loading properties into the JNDI registry in the Camel Context

```
@Override
protected JndiRegistry createRegistry() throws Exception {
  JndiRegistry registry =
         new JndiRegistry(createJndiContext());
  registry.bind("whitespace", new File("./whitespaceIndexDir"));
  registry.bind("load_dir",
        new File("src/test/resources/sources"));
  registry.bind("whitespaceAnalyzer",
        new WhitespaceAnalyzer());
  return registry;
}
...
CamelContext context = new DefaultCamelContext(createRegistry());
```

Example 2: Performing searches using a Query Producer

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
       from("direct:start").
          setHeader("QUERY", constant("Seinfeld")).
         to("lucene:searchIndex:query?analyzer=#whitespaceAnalyzer&indexDir=#whitespace&max
Hits=20").
          to("direct:next");

       from("direct:next").process(new Processor() {
          public void process(Exchange exchange) throws Exception {
             Hits hits = exchange.getIn().getBody(Hits.class);
             printResults(hits);
          }

          private void printResults(Hits hits) {
              LOG.debug("Number of hits: " + hits.getNumberOfHits());
              for (int i = 0; i < hits.getNumberOfHits(); i++) {
```

```
                LOG.debug("Hit " + i + " Index Location:" + hits.getHit().get(i).getHit
Location());
                LOG.debug("Hit " + i + " Score:" + hits.getHit().get(i).getScore());
                LOG.debug("Hit " + i + " Data:" + hits.getHit().get(i).getData());
            }
        }
    }).to("mock:searchResult");
  }
};
```

Example 3: Performing searches using a Query Processor

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        try {
            from("direct:start").
                setHeader("QUERY", constant("Rodney Dangerfield")).
              process(new LuceneQueryProcessor("target/stdindexDir", analyzer, null, 20)).

                to("direct:next");
        } catch (Exception e) {
            e.printStackTrace();
        }

        from("direct:next").process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                Hits hits = exchange.getIn().getBody(Hits.class);
                printResults(hits);
            }

            private void printResults(Hits hits) {
                LOG.debug("Number of hits: " + hits.getNumberOfHits());
                for (int i = 0; i < hits.getNumberOfHits(); i++) {
                    LOG.debug("Hit " + i + " Index Location:" + hits.getHit().get(i).getH
itLocation());
                    LOG.debug("Hit " + i + " Score:" + hits.getHit().get(i).getScore());
                    LOG.debug("Hit " + i + " Data:" + hits.getHit().get(i).getData());
                }
            }
        }).to("mock:searchResult");
  }
};
```

Chapter 54. Mail

Mail Component

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

Geronimo mail .jar

We have discovered that the geronimo mail `.jar` (v1.6) has a bug when polling mails with attachments. It cannot correctly identify the `Content-Type`. So, if you attach a `.jpeg` file to a mail and you poll it, the `Content-Type` is resolved as `text/plain` and not as `image/jpeg`. For that reason, we have added an `org.apache.camel.component.ContentTypeResolver` SPI interface which enables you to provide your own implementation and fix this bug by returning the correct Mime type based on the file name. So if the file name ends with `jpeg/jpg`, you can return `image/jpeg`.

You can set your custom resolver on the `MailComponent` instance or on the `MailEndpoint` instance. This feature is added in Camel 1.6.2/2.0.

Geronimo mail .jar

We have discovered that the geronimo mail `.jar` (v1.6) has a bug when polling mails with attachments. It cannot correctly identify the `Content-Type`. So, if you attach a `.jpeg` file to a mail and you poll it, the `Content-Type` is resolved as `text/plain` and not as `image/jpeg`. For that reason, we have added an `org.apache.camel.component.ContentTypeResolver` SPI interface which enables you to provide your own implementation and fix this bug by returning the correct Mime type based on the file name. So if the file name ends with `jpeg/jpg`, you can return `image/jpeg`.

You can set your custom resolver on the `MailComponent` instance or on the `MailEndpoint` instance. This feature is added in Fuse Mediation Router 1.6.2/2.0.

POP3 or IMAP

POP3 has some limitations and end users are encouraged to use IMAP if possible.

Using mock-mail for testing

You can use a mock framework for unit testing, which allows you to test without the need for a real mail server. However you should remember to not include the mock-mail when you go into production or other environments where you need to send mails to a real mail server. Just the presence of the mock-javamail.jar on the classpath means that it will kick in and avoid sending the mails.

URI format

Mail endpoints can have one of the following URI formats (for the protocols, SMTP, POP3, or IMAP, respectively):

```
smtp://[username@]host[:port][?options]
pop3://[username@]host[:port][?options]
imap://[username@]host[:port][?options]
```

The mail component also supports secure variants of these protocols (layered over SSL). You can enable the secure protocols by adding `s` to the scheme:

```
smtps://[username@]host[:port][?options]
pop3s://[username@]host[:port][?options]
imaps://[username@]host[:port][?options]
```

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

Sample endpoints

Typically, you specify a URI with login credentials as follows (taking SMTP as an example):

```
smtp://[username@]host[:port][?password=somepwd]
```

Alternatively, it is possible to specify both the user name and the password as query options:

```
smtp://host[:port]?password=somepwd&username=someuser
```

For example:

```
smtp://mycompany.mailserver:30?password=tiger&username=scott
```

Default ports

As of Fuse Mediation Router 1.4, default port numbers are supported. If the port number is omitted, Fuse Mediation Router determines the port number to use based on the protocol.

Protocol	Default Port Number
SMTP	25
SMTPS	465
POP3	110
POP3S	995
IMAP	143
IMAPS	993

Options

Property	Default	Description
host		The host name or IP address to connect to.
port	See #DefaultPorts	The TCP port number to connect on.
username		The user name on the email server.
password	null	The password on the email server.
ignoreUriScheme	false	If false, Fuse Mediation Router uses the scheme to deter protocol (POP, IMAP, SMTP etc.)
defaultEncoding	null	The default encoding to use for Mime Messages.
contentType	text/plain	New option in **Fuse Mediation Router 1.5**. The mail mes Use text/html for HTML mails.
folderName	INBOX	The folder to poll.
destination	username@host	@**deprecated** Use the to option instead. The TO recipien email).
to	username@host	As of **Fuse Mediation Router 1.4**, the TO recipients (the mail). Separate multiple email addresses with a comma.
CC	null	As of **Fuse Mediation Router 1.4**, the CC recipients (the mail). Separate multiple email addresses with a comma.
BCC	null	As of **Fuse Mediation Router 1.4**, the BCC recipients (th mail). Separate multiple email addresses with a comma.
from	camel@localhost	The FROM email address.
subject		As of **Camel 2.3**, the Subject of the message being sent. subject in the header takes precedence over this option.
deleteProcessedMessages	true	Deletes the messages after they have been processed. Thi the DELETED flag on the mail message. If false, the SEEN

		As of **Fuse Mediation Router 1.5**, the default setting is `fals` is named `delete` in Camel 2.0 onwards.
`delete`	`false`	**Fuse Mediation Router 2.0**: Deletes the messages after the processed. This is done by setting the `DELETED` flag on the m `false`, the `SEEN` flag is set instead.
`processOnlyUnseenMessages`	`false`	As of **Fuse Mediation Router 1.4**, it is possible to configure endpoint so that it processes only unseen messages (that is, r or all messages. Note that Fuse Mediation Router always sk messages. Setting this option to `true` will filter to only unseen of **Fuse Mediation Router 1.5**, the default setting is `true`. P support the `SEEN` flag, so this option is not supported in POP instead. This option is named `unseen` in Camel 2.0 onwards
`unseen`	`true`	**Fuse Mediation Router 2.0**: Is used to fetch only unseen m is, new messages). Note that POP3 does not support the `SEEN` instead.
`fetchSize`	`-1`	As of **Fuse Mediation Router 1.4**, this option sets the maxir messages to consume during a poll. This can be used to avo a mail server, if a mailbox folder contains a lot of messages. of `-1` means no fetch size and all messages will be consume value to 0 is a special corner case, where Fuse Mediation R consume any messages at all.
`alternateBodyHeader`	`mail_alternateBody`	**Fuse Mediation Router 1.6.1**: Specifies the key to an IN me that contains an alternative email body. For example, if you s `text/html` format and want to provide an alternative mail body email clients, set the alternative mail body with this key as a h Mediation Router 2.0, this option has been renamed to `alternativeBodyHeader`.
`alternativeBodyHeader`	`CamelMailAlternativeBody`	**Fuse Mediation Router 2.0**: Specifies the key to an IN messa contains an alternative email body. For example, if you send `text/html` format and want to provide an alternative mail body email clients, set the alternative mail body with this key as a
`debugMode`	`false`	As of **Fuse Mediation Router 1.4**, it is possible to enable de the underlying mail framework. The SUN Mail framework log messages to `System.out` by default.
`connectionTimeout`	`30000`	As of **Fuse Mediation Router 1.4**, the connection timeout ca in milliseconds. Default is 30 seconds.
`consumer.initialDelay`	`1000`	Milliseconds before the polling starts.
`consumer.delay`	`60000`	As of **Fuse Mediation Router 1.4**, the default consumer del seconds. Fuse Mediation Router will therefore only poll the r

		minute to avoid overloading the mail server. The default v Mediation Router 1.3 is 500 milliseconds.
`consumer.useFixedDelay`	`false`	Set to `true` to use a fixed delay between polls, otherwise See ScheduledExecutorService[1] in JDK for details.
`mail.XXX`	`null`	As of **Fuse Mediation Router 2.0**, you can set any additi properties[2]. For instance if you want to set a special prop POP3 you can now provide the option directly in the URI `mail.pop3.forgettopheaders=true`. You can set multip for example: `mail.pop3.forgettopheaders=true&mail.mime.encod`
`maxMessagesPerPoll`	`0`	**Fuse Mediation Router 2.0:** Specifies the maximum nur to gather per poll. By default, no maximum is set. Can be of e.g. 1000 to avoid downloading thousands of files whe up. Set a value of 0 or negative to disable this option.
`javaMailSender`	`null`	**Fuse Mediation Router 2.0:** Specifies a pluggable `org.springframework.mail.javamail.JavaMailSende` to use a custom email implementation. If none provided, Router uses the default, `org.springframework.mail.javamail.JavaMailSende`
`ignoreUnsupportedCharset`	`false`	**Fuse Mediation Router 2.0:** Option to let Fuse Mediation unsupported charset in the local JVM when sending mails unsupported then `charset=XXX` (where `XXX` represents th charset) is removed from the `content-type` and it relies default instead.

SSL support

The underlying mail framework is responsible for providing SSL support. Fuse Mediation Router uses SUN JavaMail, which only trusts certificates issued by well known Certificate Authorities. So if you issue your own certificate, you have to import it into the local Java keystore file (see `SSLNOTES.txt` in JavaMail for details).

Defaults changed in Fuse Mediation Router 1.4

As of Fuse Mediation Router 1.4 the default consumer delay is now 60 seconds. Fuse Mediation Router will therefore only poll the mailbox once a minute to avoid overloading the mail server. The default value in Fuse Mediation Router 1.3 is 500 milliseconds.

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html
[2] http://java.sun.com/products/javamail/javadocs/index.html

Defaults changed in Fuse Mediation Router 1.5

In Fuse Mediation Router 1.5 the following default options have changed:

- `deleteProcessedMessages` is now **false**, as we felt Fuse Mediation Router should not delete mails on the mail server by default.
- `processOnlyUnseenMessages` is now **true**, as we felt Fuse Mediation Router should only poll new mails by default.

Mail Message Content

Fuse Mediation Router uses the message exchange's IN body as the MimeMessage[3] text content. The body is converted to `String.class`.

Fuse Mediation Router copies all of the exchange's IN headers to the MimeMessage[4] headers.

The subject of the MimeMessage[5] can be configured using a header property on the IN message. The code below demonstrates this:

```
from("direct:a").setHeader("subject", constant(subject)).to("smtp://james2@localhost");
```

The same applies for other MimeMessage headers such as recipients, so you can use a header property as `To`:

```
Map map = new HashMap();
map.put("To", "davsclaus@apache.org");
map.put("From", "jstrachan@apache.org");
map.put("Subject", "Camel rocks");

String body = "Hello Claus.\nYes it does.\n\nRegards James.";
template.sendBodyAndHeaders("smtp://davsclaus@apache.org", body, map);
```

Headers take precedence over pre-configured recipients

From Fuse Mediation Router 1.5 onwards, the recipients specified in the message headers always take precedence over recipients pre-configured in the endpoint URI. The idea is that if you provide any recipients in the message headers, that is what you get. The recipients pre-configured in the endpoint URI are treated as a fallback.

[3] http://java.sun.com/javaee/5/docs/api/javax/mail/internet/MimeMessage.html
[4] http://java.sun.com/javaee/5/docs/api/javax/mail/internet/MimeMessage.html
[5] http://java.sun.com/javaee/5/docs/api/javax/mail/internet/MimeMessage.html

In the sample code below, the email message is sent to `davsclaus@apache.org`, because it takes precedence over the pre-configured recipient, `info@mycompany.com`. Any `CC` and `BCC` settings in the endpoint URI are also ignored and those recipients will not receive any mail. The choice between headers and pre-configured settings is all or nothing: the mail component *either* takes the recipients exclusively from the headers or exclusively from the pre-configured settings. It is not possible to mix and match headers and pre-configured settings.

```
        Map<String, Object> headers = new HashMap<String, Object>();
        headers.put("to", "davsclaus@apache.org");

        template.sendBodyAndHeaders("smtp://admin@localhost?to=info@mycompany.com", "Hello
 World", headers);
```

Multiple recipients for easier configuration

As of Fuse Mediation Router 1.5, it is possible to set multiple recipients using a comma-separated or a semicolon-separated list. This applies both to header settings and to settings in an endpoint URI. For example:

```
        Map<String, Object> headers = new HashMap<String, Object>();
        headers.put("to", "davsclaus@apache.org ; jstrachan@apache.org ; ningji
ang@apache.org");
```

The preceding example uses a semicolon, `;`, as the separator character.

Setting sender name and email

You can specify recipients in the format, `name <email>`, to include both the name and the email address of the recipient.

For example, you define the following headers on the a Message:

```
Map headers = new HashMap();
map.put("To", "Claus Ibsen <davsclaus@apache.org>");
map.put("From", "James Strachan <jstrachan@apache.org>");
map.put("Subject", "Camel is cool");
```

SUN JavaMail

SUN JavaMail[6] is used under the hood for consuming and producing mails. We encourage end-users to consult these references when using either POP3 or IMAP protocol. Note particularly that POP3 has a much more limited set of features than IMAP.

- SUN POP3 API[7]

[6] http://java.sun.com/products/javamail/
[7] http://java.sun.com/products/javamail/javadocs/com/sun/mail/pop3/package-summary.html

- SUN IMAP API[8]
- And generally about the MAIL Flags[9]

Samples

We start with a simple route that sends the messages received from a JMS queue as emails. The email account is the `admin` account on `mymailserver.com`.

```
from("jms://queue:subscription").to("smtp://admin@mymailserver.com?password=secret");
```

In the next sample, we poll a mailbox for new emails once every minute. Notice that we use the special `consumer` option for setting the poll interval, `consumer.delay`, as 60000 milliseconds = 60 seconds.

```
from("imap://admin@mymailserver.com?password=secret&processOnlyUnseenMessages=true&con
sumer.delay=60000").to("seda://mails");
```

In this sample we want to send a mail to multiple recipients. This feature was introduced in camel 1.4:

```
// all the recipients of this mail are:
// To: camel@riders.org , easy@riders.org
// CC: me@you.org
// BCC: someone@somewhere.org
String recipients = "&To=camel@riders.org,easy@riders.org&CC=me@you.org&BCC=someone@some
where.org";

from("direct:a").to("smtp://you@mymailserver.com?password=secret&From=you@apache.org" +
recipients);
```

Sending mail with attachment sample

Attachments are not support by all Fuse Mediation Router components

The *Attachments API* is based on the Java Activation Framework and is generally only used by the Mail API. Since many of the other Fuse Mediation Router components do not support attachments, the attachments could potentially be lost as they propagate along the route. The rule of thumb, therefore, is to add attachments just before sending a message to the mail endpoint.

[8] http://java.sun.com/products/javamail/javadocs/com/sun/mail/imap/package-summary.html
[9] http://java.sun.com/products/javamail/javadocs/javax/mail/Flags.html

The mail component supports attachments, which is a feature that was introduced in Fuse Mediation Router 1.4. In the sample below, we send a mail message containing a plain text message with a logo file attachment.

```
// create an exchange with a normal body and attachment to be produced as email
Endpoint endpoint = context.getEndpoint("smtp://james@mymailserver.com?password=secret");

// create the exchange with the mail message that is multipart with a file and a Hello World
 text/plain message.
Exchange exchange = endpoint.createExchange();
Message in = exchange.getIn();
in.setBody("Hello World");
in.addAttachment("logo.jpeg", new DataHandler(new FileDataSource("src/test/data/logo.jpeg")));

// create a producer that can produce the exchange (= send the mail)
Producer producer = endpoint.createProducer();
// start the producer
producer.start();
// and let it go (processes the exchange by sending the email)
producer.process(exchange);
```

SSL sample

In this sample, we want to poll our Google mail inbox for mails. To download mail onto a local mail client, Google mail requires you to enable and configure SSL. This is done by logging into your Google mail account and changing your settings to allow IMAP access. Google have extensive documentation on how to do this.

```
from("imaps://imap.gmail.com?username=YOUR_USERNAME@gmail.com&password=YOUR_PASSWORD"
    + "&deleteProcessedMessages=false&processOnlyUnseenMessages=true&con
sumer.delay=60000").to("log:newmail");
```

The preceding route polls the Google mail inbox for new mails once every minute and logs the received messages to the `newmail` logger category. Running the sample with `DEBUG` logging enabled, we can monitor the progress in the logs:

```
2008-05-08 06:32:09,640 DEBUG MailConsumer - Connecting to MailStore imaps//imap.gmail.com:993
 (SSL enabled), folder=INBOX
2008-05-08 06:32:11,203 DEBUG MailConsumer - Polling mailfolder: imaps//imap.gmail.com:993
 (SSL enabled), folder=INBOX
2008-05-08 06:32:11,640 DEBUG MailConsumer - Fetching 1 messages. Total 1 messages.
2008-05-08 06:32:12,171 DEBUG MailConsumer - Processing message: messageNumber=[332],
from=[James Bond <007@mi5.co.uk>], to=YOUR_USERNAME@gmail.com], subject=[...
2008-05-08 06:32:12,187 INFO  newmail - Exchange[MailMessage: messageNumber=[332], from=[James
 Bond <007@mi5.co.uk>], to=YOUR_USERNAME@gmail.com], subject=[...
```

Consuming mails with attachment sample

In this sample we poll a mailbox and store all attachments from the mails as files. First, we define a route to poll the mailbox. As this sample is based on google mail, it uses the same route as shown in the SSL sample:

```
from("imaps://imap.gmail.com?username=YOUR_USERNAME@gmail.com&password=YOUR_PASSWORD"
    + "&deleteProcessedMessages=false&processOnlyUnseenMessages=true&con
sumer.delay=60000").process(new MyMailProcessor());
```

Instead of logging the mail we use a processor where we can process the mail from java code:

```
    public void process(Exchange exchange) throws Exception {
        // the API is a bit clunky so we need to loop
        Map<String, DataHandler> attachments = exchange.getIn().getAttachments();
        if (attacments.size() > 0) {
            for (String name : attachments.keySet()) {
                DataHandler dh = attachments.get(name);
                // get the file name
                String filename = dh.getName();

                // get the content and convert it to byte[]
                 byte[] data = exchange.getContext().getTypeConverter().con
vertTo(byte[].class, dh.getInputStream());

                // write the data to a file
                FileOutputStream out = new FileOutputStream(filename);
                out.write(data);
                out.flush();
                out.close();
            }
        }
   }
```

As you can see the API to handle attachments is a bit clunky but it's there so you can get the `javax.activation.DataHandler` so you can handle the attachments using standard API.

Chapter 55. MINA

MINA Component

The **mina:** component is a transport for working with Apache MINA[1]

URI format

```
mina:tcp://hostname[:port][?options]
mina:udp://hostname[:port][?options]
mina:vm://hostname[:port][?options]
```

From Fuse Mediation Router 1.3 onwards you can specify a codec in the Registry using the **codec** option. If you are using TCP and no codec is specified then the `textline` flag is used to determine if text line based codec or object serialization should be used instead. By default the object serialization is used.

For UDP, if no codec is specified the default uses a basic `ByteBuffer` based codec.

The VM protocol is used as a direct forwarding mechanism in the same JVM. See the MINA VM-Pipe API documentation[2] for details.

A Mina producer has a default timeout value of 30 seconds, while it waits for a response from the remote server.

In normal use, `camel-mina` only supports marshalling the body content—essage headers and exchange properties are not sent. However, the option, **transferExchange**, does allow you to transfer the exchange itself over the wire. See options below.

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

Options

Option	Default Value	Description
`codec`	`null`	As of 1.3, you can refer to a named `ProtocolCodecFactory` instance in your Registry such as your Spring `ApplicationContext`, which is then used for the marshalling.

[1] http://mina.apache.org/
[2] http://mina.apache.org/report/1.1/apidocs/org/apache/mina/transport/vmpipe/package-summary.html

`codec`	`null`	**Fuse Mediation Router 2.0:** You must use the `#` notation to look up your codec in the Registry. For example, use `#myCodec` to look up a bean with the `id` value, `myCodec`.
`disconnect`	`false`	**Camel 2.3:** Whether or not to disconnect(close) from Mina session right after use. Can be used for both consumer and producer.
`textline`	`false`	Only used for TCP. If no codec is specified, you can use this flag in 1.3 or later to indicate a text line based codec; if not specified or the value is `false`, then Object Serialization is assumed over TCP.
`textlineDelimiter`	`DEFAULT`	**Fuse Mediation Router 1.6.0/2.0** Only used for TCP and if **textline=true**. Sets the text line delimiter to use. Possible values are: `DEFAULT`, `AUTO`, `WINDOWS`, `UNIX` or `MAC`. If none provided, Fuse Mediation Router will use `DEFAULT`. This delimiter is used to mark the end of text.
`sync`	`true`	As of 1.3, you can configure the exchange pattern to be either InOnly (default) or InOut. Setting `sync=true` means a synchronous exchange (InOut), where the client can read the response from MINA (the exchange Out message). The default value has changed in Fuse Mediation Router 1.5 to `true`. In older releases, the default value is `false`.
`lazySessionCreation`	*See description*	As of 1.3, sessions can be lazily created to avoid exceptions, if the remote server is not up and running when the Fuse Mediation Router producer is started. From Fuse Mediation Router 2.0 onwards, the default is `true`. In Fuse Mediation Router 1.x, the default is `false`.
`timeout`	`30000`	As of 1.3, you can configure the timeout that specifies how long to wait for a response from a remote server. The timeout unit is in milliseconds, so 60000 is 60 seconds. The timeout is only used for Mina producer.
`encoding`	*JVM Default*	As of 1.3, you can configure the encoding (a charset name[3]) to use for the TCP textline codec and the UDP protocol. If not provided, Fuse Mediation Router will use the JVM default Charset[4].
`transferExchange`	`false`	Only used for TCP. As of 1.3, you can transfer the exchange over the wire instead of just the body. The following fields are transferred: In body, Out body, fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are *serializable*. Fuse Mediation Router will exclude any non-serializable objects and log it at `WARN` level.
`minaLogger`	`false`	As of 1.3, you can enable the Apache MINA logging filter. Apache MINA uses `slf4j` logging at `INFO` level to log all input and output.

[3] http://java.sun.com/j2se/1.5.0/docs/api/java/nio/charset/Charset.html
[4] http://java.sun.com/j2se/1.5.0/docs/api/java/nio/charset/Charset.html#defaultCharset()

`filters`	`null`	As of 2.0, you can set a list of Mina IoFilters[5] to register. The `filters` value must be one of the following: • **Camel 2.2:** comma-separated list of bean references (e.g. `#filterBean1,#filterBean2`) where each bean must be of type `org.apache.mina.common.IoFilter`. • **Camel 2.0:** a reference to a bean of type `List<org.apache.mina.common.IoFilter>`.
`encoderMaxLineLength`	`-1`	As of 2.1, you can set the textline protocol encoder max line length. By default the default value of Mina itself is used which are `Integer.MAX_VALUE`.
`decoderMaxLineLength`	`-1`	As of 2.1, you can set the textline protocol decoder max line length. By default the default value of Mina itself is used which are 1024.
`allowDefaultCodec`	`true`	The mina component installs a default codec if both, `codec` is `null` and `textline` is `false`. Setting `allowDefaultCodec` to `false` prevents the mina component from installing a default codec as the first element in the filter chain. This is useful in scenarios where another filter must be the first in the filter chain, like the SSL filter.
`disconnectOnNoReply`	`true`	**Camel 2.3:** If sync is enabled then this option dictates MinaConsumer if it should disconnect where there is no reply to send back.
`noReplyLogLevel`	`WARN`	**Camel 2.3:** If sync is enabled this option dictates MinaConsumer which logging level to use when logging a there is no reply to send back. Values are: `FATAL, ERROR, INFO, DEBUG, OFF`.

Default behavior changed

In Fuse Mediation Router 2.0 the **codec** option must use `#` notation for lookup of the codec bean in the Registry.
In Fuse Mediation Router 2.0 the **lazySessionCreation** option now defaults to `true`.

In Fuse Mediation Router 1.5 the `sync` option has changed its default value from `false` to `true`, as we felt it was confusing for end-users when they used Mina to call remote servers and Fuse Mediation Router wouldn't wait for the response.

In Fuse Mediation Router 1.4 or later `codec=textline` is no longer supported. Use the `textline=true` option instead.

[5] http://mina.apache.org/iofilter.html

Using a custom codec

See the Mina documentation[6] how to write your own codec. To use your custom codec with `camel-mina`, you should register your codec in the Registry; for example, by creating a bean in the Spring XML file. Then use the `codec` option to specify the bean ID of your codec. See HL7 on page 219 that has a custom codec.

Sample with sync=false

In this sample, Fuse Mediation Router exposes a service that listens for TCP connections on port 6200. We use the **textline** codec. In our route, we create a Mina consumer endpoint that listens on port 6200:

```
from("mina:tcp://localhost:6200?textline=true&sync=false").to("mock:result");
```

As the sample is part of a unit test, we test it by sending some data to it on port 6200.

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedBodiesReceived("Hello World");

template.sendBody("mina:tcp://localhost:6200?textline=true&sync=false", "Hello World");

assertMockEndpointsSatisfied();
```

Sample with sync=true

In the next sample, we have a more common use case where we expose a TCP service on port 6201 also use the textline codec. However, this time we want to return a response, so we set the `sync` option to `true` on the consumer.

```
from("mina:tcp://localhost:6201?textline=true&sync=true").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        exchange.getOut().setBody("Bye " + body);
    }
});
```

Then we test the sample by sending some data and retrieving the response using the `template.requestBody()` method. As we know the response is a `String`, we cast it to `String` and can assert that the response is, in fact, something we have dynamically set in our processor code logic.

```
String response = (String)template.requestBody("mina:tcp://local
host:6201?textline=true&sync=true", "World");
assertEquals("Bye World", response);
```

[6] http://mina.apache.org/tutorial-on-protocolcodecfilter.html

Sample with Spring DSL

Spring DSL can, of course, also be used for Mina. In the sample below we expose a TCP server on port 5555:

```
   <route>
     <from uri="mina:tcp://localhost:5555?textline=true"/>
     <to uri="bean:myTCPOrderHandler"/>
  </route>
```

In the route above, we expose a TCP server on port 5555 using the textline codec. We let the Spring bean with ID, `myTCPOrderHandler`, handle the request and return a reply. For instance, the handler bean could be implemented as follows:

```
    public String handleOrder(String payload) {
        ...
        return "Order: OK"
   }
```

Configuring Mina endpoints using Spring bean style

Available as of Fuse Mediation Router 2.0

Configuration of Mina endpoints is now possible using regular Spring bean style configuration in the Spring DSL.

However, in the underlying Apache Mina toolkit, it is relatively difficult to set up the acceptor and the connector, because you can *not* use simple setters. To resolve this difficulty, we leverage the `MinaComponent` as a Spring factory bean to configure this for us. If you really need to configure this yourself, there are setters on the `MinaEndpoint` to set these when needed.

The sample below shows the factory approach:

```
<!-- Creating mina endpoints is a bit complex so we reuse MinaComponnet
     as a factory bean to create our endpoint, this is the easiest to do -->
<bean id="myMinaFactory" class="org.apache.camel.component.mina.MinaComponent">
    <!-- we must provide a camel context so we refer to it by its id -->
    <constructor-arg index="0" ref="myCamel"/>
</bean>

<!-- This is our mina endpoint configured with spring, we will use the factory above
     to create it for us. The goal is to invoke the createEndpoint method with the
     mina configuration parameter we defined using the constructor-arg option -->
<bean id="myMinaEndpoint"
      factory-bean="myMinaFactory"
      factory-method="createEndpoint">
    <!-- and here we can pass it our configuration -->
```

```
    <constructor-arg index="0" ref="myMinaConfig"/>
</bean>

<!-- this is our mina configuration with plain properties -->
<bean id="myMinaConfig" class="org.apache.camel.component.mina.MinaConfiguration">
    <property name="protocol" value="tcp"/>
    <property name="host" value="localhost"/>
    <property name="port" value="1234"/>
    <property name="sync" value="false"/>
</bean>
```

And then we can refer to our endpoint directly in the route, as follows:

```
<route>
    <!-- here we route from or mina endpoint we have defined above -->
    <from ref="myMinaEndpoint"/>
    <to uri="mock:result"/>
</route>
```

Closing Session When Complete

Available as of Fuse Mediation Router 1.6.1

When acting as a server you sometimes want to close the session when, for example, a client conversion is finished. To instruct Fuse Mediation Router to close the session, you should add a header with the key `CamelMinaCloseSessionWhenComplete` set to a boolean `true` value.

For instance, the example below will close the session after it has written the `bye` message back to the client:

```
        from("mina:tcp://localhost:8080?sync=true&textline=true").process(new Processor()
{
            public void process(Exchange exchange) throws Exception {
                String body = exchange.getIn().getBody(String.class);
                exchange.getOut().setBody("Bye " + body);
              exchange.getOut().setHeader(MinaConsumer.HEADER_CLOSE_SESSION_WHEN_COMPLETE,
 true);
            }
        });
```

Get the IoSession for message

Available since Fuse Mediation Router 2.1 You can get the IoSession from the message header with this key `MinaEndpoint.HEADER_MINA_IOSESSION`, and also get the local host address with the key `MinaEndpoint.HEADER_LOCAL_ADDRESS` and remote host address with the key `MinaEndpoint.HEADER_REMOTE_ADDRESS`.

Configuring Mina filters

Available since Fuse Mediation Router 2.0

Filters permit you to use some Mina Filters, such as `SslFilter`. You can also implement some customized filters. Please note that `codec` and `logger` are also implemented as Mina filters of type, `IoFilter`. Any filters you may define are appended to the end of the filter chain; that is, after `codec` and `logger`.

For instance, the example below will send a keep-alive message after 10 seconds of inactivity:

```
public class KeepAliveFilter extends IoFilterAdapter {
    @Override
    public void sessionCreated(NextFilter nextFilter, IoSession session)
            throws Exception {
        session.setIdleTime(IdleStatus.BOTH_IDLE, 10);

        nextFilter.sessionCreated(session);
    }

    @Override
    public void sessionIdle(NextFilter nextFilter, IoSession session,
            IdleStatus status) throws Exception {
        session.write("NOOP"); // NOOP is a FTP command for keep alive
        nextFilter.sessionIdle(session, status);
    }
}
```

As Fuse Mediation Router Mina may use a request-reply scheme, the endpoint as a client would like to drop some message, such as greeting when the connection is established. For example, when you connect to an FTP server, you will get a `220` message with a greeting (`220 Welcome to Pure-FTPd`). If you don't drop the message, your request-reply scheme will be broken.

```
public class DropGreetingFilter extends IoFilterAdapter {

    @Override
    public void messageReceived(NextFilter nextFilter, IoSession session,
            Object message) throws Exception {
        if (message instanceof String) {
            String ftpMessage = (String) message;
           // "220" is given as greeting. "200 Zzz" is given as a response to "NOOP" (keep
 alive)
            if (ftpMessage.startsWith("220") || or ftpMessage.startsWith("200 Zzz")) {
                // Dropping greeting
                return;
            }
        }
        nextFilter.messageReceived(session, message);
```

```
    }
}
```

Then, you can configure your endpoint using Spring DSL:

```
<bean id="myMinaFactory" class="org.apache.camel.component.mina.MinaComponent">
    <constructor-arg index="0" ref="camelContext" />
</bean>

<bean id="myMinaEndpoint"
      factory-bean="myMinaFactory"
      factory-method="createEndpoint">
    <constructor-arg index="0" ref="myMinaConfig"/>
</bean>

<bean id="myMinaConfig" class="org.apache.camel.component.mina.MinaConfiguration">
    <property name="protocol" value="tcp" />
    <property name="host" value="localhost" />
    <property name="port" value="2121" />
    <property name="sync" value="true" />
    <property name="minaLogger" value="true" />
    <property name="filters" ref="listFilters"/>
</bean>

<bean id="listFilters" class="java.util.ArrayList" >
    <constructor-arg>
        <list value-type="org.apache.mina.common.IoFilter">
            <bean class="com.example.KeepAliveFilter"/>
            <bean class="com.example.DropGreetingFilter"/>
        </list>
    </constructor-arg>
</bean>
```

Chapter 56. Mock

Mock Component

The Mock component provides a powerful declarative testing mechanism, which is similar to jMock[1] in that it allows declarative expectations to be created on any Mock endpoint before a test begins. Then the test is run, which typically fires messages to one or more endpoints, and finally the expectations can be asserted in a test case to ensure the system worked as expected.

This allows you to test various things like:

- The correct number of messages are received on each endpoint,
- The correct payloads are received, in the right order,
- Messages arrive on an endpoint in order, using some Expression to create an order testing function,
- Messages arrive match some kind of Predicate such as that specific headers have certain values, or that parts of the messages match some predicate, such as by evaluating an XPath or XQuery Expression.

Note that there is also the Test endpoint on page 505 which is a Mock endpoint, but which uses a second endpoint to provide the list of expected message bodies and automatically sets up the Mock endpoint assertions. In other words, it's a Mock endpoint that automatically sets up its assertions from some sample messages in a File or database on page 315, for example.

URI format

```
mock:someName[?options]
```

Where **someName** can be any string that uniquely identifies the endpoint.

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

Options

Option	Default	Description
`reportGroup`	`null`	A size to use a throughput logger on page 331 for reporting

[1] http://jmock.org

Simple Example

Here's a simple example of Mock endpoint in use. First, the endpoint is resolved on the context. Then we set an expectation, and then, after the test has run, we assert that our expectations have been met.

```
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 typically always call the assertIsSatisfied() method[2] to test that the expectations were met after running a test.

Fuse Mediation Router will by default wait 20 seconds when the `assertIsSatisfied()` is invoked. This can be configured by setting the `setResultWaitTime(millis)` method.

Setting expectations

You can see from the javadoc of MockEndpoint[3] the various helper methods you can use to set expectations. The main methods are as follows:

Method	Description
expectedMessageCount(int)[4]	To define the expected message count on the endpoint.
expectedMinimumMessageCount(int)[5]	To define the minimum number of expected messages on the endpoint.
expectedBodiesReceived(...)[6]	To define the expected bodies that should be received (in order).
expectedHeaderReceived(...)[7]	To define the expected header that should be received

[2] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html#assertIsSatisfied()
[3] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html
[4] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html#expectedMessageCount(int)
[5] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html#expectedMinimumMessageCount(int)
[6] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html#expectedBodiesReceived(java.lang.Object...)
[7] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html#expectedHeaderReceived(java.lang.String,%20java.lang.String)

expectsAscending(Expression)[8]	To add an expectation that messages are received in order, using the given Expression to compare messages.
expectsDescending(Expression)[9]	To add an expectation that messages are received in order, using the given Expression to compare messages.
expectsNoDuplicates(Expression)[10]	To add an expectation that no duplicate messages are received; using an Expression to calculate a unique identifier for each message. This could be something like the `JMSMessageID` if using JMS, or some unique reference number within the message.

Here's another example:

```
resultEndpoint.expectedBodiesReceived("firstMessageBody", "secondMessageBody", "thirdMes
sageBody");
```

Adding expectations to specific messages

In addition, you can use the message(int messageIndex)[11] method to add assertions about a specific message that is received.

For example, to add expectations of the headers or body of the first message (using zero-based indexing like `java.util.List`), you can use the following code:

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

There are some examples of the Mock endpoint in use in the camel-core processor tests[12].

[8] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html#expectsAscending(org.apache.camel.Expression)
[9] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html#expectsDescending(org.apache.camel.Expression)
[10] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html#expectsNoDuplicates(org.apache.camel.Expression)
[11] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html#message(int)
[12] http://svn.apache.org/viewvc/camel/trunk/camel-core/src/test/java/org/apache/camel/processor/

Chapter 57. MSV

MSV Component

The MSV component performs XML validation of the message body using the MSV Library[1] and any of the supported XML schema languages, such as XML Schema[2] or RelaxNG XML Syntax[3].

Note that the Jing on page 291 component also supports RelaxNG Compact Syntax[4]

URI format

```
msv:someLocalOrRemoteResource[?options]
```

Where **someLocalOrRemoteResource** is some URL to a local resource on the classpath or a full URL to a remote resource or resource on the file system. For example

```
msv:org/foo/bar.rng
msv:file:../foo/bar.rng
msv:http://acme.com/cheese.rng
```

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

Options

Option	Default	Description
`useDom`	`true`	**Fuse Mediation Router 2.0:** Whether DOMSource/DOMResult or SaxSource/SaxResult should be used by the validator. **Note:** DOM must be used by the MSV on page 361 component.

Example

The following example[5] shows how to configure a route from endpoint **direct:start** which then goes to one of two endpoints, either **mock:valid** or **mock:invalid** based on whether or not the XML matches the given RelaxNG XML Schema[6] (which is supplied on the classpath).

[1] https://msv.dev.java.net/
[2] http://www.w3.org/XML/Schema
[3] http://relaxng.org/
[4] http://relaxng.org/compact-tutorial-20030326.html
[5] http://svn.apache.org/repos/asf/camel/trunk/components/camel-msv/src/test/resources/org/apache/camel/component/validator/msv/camelContext.xml
[6] http://relaxng.org/

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <doTry>
            <to uri="msv:org/apache/camel/component/validator/msv/schema.rng"/>
            <to uri="mock:valid"/>

            <doCatch>
                <exception>org.apache.camel.ValidationException</exception>
                <to uri="mock:invalid"/>
            </doCatch>
            <doFinally>
                <to uri="mock:finally"/>
            </doFinally>
        </doTry>
    </route>
</camelContext>
```

Chapter 58. Nagios

Nagios

Available as of Fuse Mediation Router 2.3

The Nagios on page 363 component allows you to send passive checks to Nagios[1].

URI format

```
nagios://host[:port][?Options]
```

Fuse Mediation Router provides two abilities with the Nagios on page 363 component. You can send passive check messages by sending a message to its endpoint. Fuse Mediation Router also provides a EventNotifer[2] which allows you to send notifications to Nagios.

Options

Name	Default Value	Description
`host`	none	This is the address of the Nagios on page 363 host where checks should be send.
`port`		The port number of the host.
`password`		Password to be authenticated when sending checks to Nagios.
`connectionTimeout`	5000	Connection timeout in millis.
`timeout`	5000	Sending timeout in millis.
`nagiosSettings`		To use an already configured `com.googlecode.jsendnsca.core.NagiosSettings` object.
`sendSync`	`true`	Whether or not to use synchronous when sending a passive check. Setting it to `false` will allow Fuse Mediation Router to continue routing the message and the passive check message will be send asynchronously.

Headers

Name	Description

[1] http://nagios.org
[2] Camel JMX

`CamelNagiosHostName`	This is the address of the Nagios on page 363 host where checks should be send. This header will override any existing hostname configured on the endpoint.
`CamelNagiosLevel`	This is the severity level. You can use values `CRITICAL, WARNING, OK`. Fuse Mediation Router will by default use `OK`.
`CamelNagiosServiceName`	The servie name. Will default use the CamelContext name.

Sending message examples

You can send a message to Nagios where the message payload contains the message. By default it will be `OK` level and use the CamelContext name as the service name. You can overrule these values using headers as shown above.

For example we send the `Hello Nagios` message to Nagios as follows:

```
    template.sendBody("direct:start", "Hello Nagios");

    from("direct:start").to("nagios:127.0.0.1:5667?password=secret").to("mock:result");
```

To send a `CRITICAL` message you can send the headers such as:

```
        Map headers = new HashMap();
        headers.put(NagiosConstants.LEVEL, "CRITICAL");
        headers.put(NagiosConstants.HOST_NAME, "myHost");
        headers.put(NagiosConstants.SERVICE_NAME, "myService");
        template.sendBodyAndHeaders("direct:start", "Hello Nagios", headers);
```

Using NagiosEventNotifer

The Nagios on page 363 component also provides an EventNotifer[3] which you can use to send events to Nagios. For example we can enable this from Java as follows:

```
        NagiosEventNotifier notifier = new NagiosEventNotifier();
        notifier.getConfiguration().setHost("localhost");
        notifier.getConfiguration().setPort(5667);
        notifier.getConfiguration().setPassword("password");

        CamelContext context = ...
        context.getManagementStrategy().addEventNotifier(notifier);
        return context;
```

[3] Camel JMX

In Spring XML its just a matter of defining a Spring bean with the type `EventNotifier` and Fuse Mediation Router will pick it up as documented here: Advanced configuration of CamelContext using Spring[4].

[4] Advanced configuration of CamelContext using Spring

Chapter 59. Netty

Netty Component

Available as of Fuse Mediation Router 2.3

The **netty** component in Fuse Mediation Router is a socket communication component, based on the JBoss Netty community offering (available under an Apache 2.0 license). Netty is a NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients. Netty greatly simplifies and streamlines network programming such as TCP and UDP socket server.

This Fuse Mediation Router component supports both producer and consumer endpoints.

The netty component has several options and allows fine-grained control of a number of TCP/UDP communication parameters (buffer sizes, keepAlives, tcpNoDelay etc) and facilitates both In-Only and In-Out communication on a Fuse Mediation Router route.

URI format

The URI scheme for a netty component is as follows

```
netty:tcp://localhost:99999[?options]
netty:udp://remotehost:99999/[?options]
```

This component supports producer and consumer endpoints for both TCP and UDP.

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

Options

Name	Default Value	Description
`keepAlive`	`true`	Setting to ensure socket is not closed due to inactivity
`tcpNoDelay`	`true`	Setting to improve TCP protocol performance
`broadcast`	`false`	Setting to choose Multicast over UDP
`connectTimeout`	`10000`	Time to wait for a socket connection to be available. Value is in millis.
`reuseAddress`	`true`	Setting to facilitate socket multiplexing
`sync`	`true`	Setting to set endpoint as one-way or request-response
`ssl`	`false`	Setting to specify whether SSL encryption is applied to this endpoint

sendBufferSize	65536 bytes	The TCP/UDP buffer sizes to be used during outbound communication. Size is bytes.
receiveBufferSize	65536 bytes	The TCP/UDP buffer sizes to be used during inbound communication. Size is bytes.
corePoolSize	10	The number of allocated threads at component startup. Defaults to 10
maxPoolSize	100	The maximum number of threads that may be allocated to this endpoint. Defaults to 100
disconnect	false	Whether or not to disconnect(close) from Netty Channel right after use. Can be used for both consumer and producer.
lazyChannelCreation	true	Channels can be lazily created to avoid exceptions, if the remote server is not up and running when the Fuse Mediation Router producer is started.
transferExchange	false	Only used for TCP. You can transfer the exchange over the wire instead of just the body. The following fields are transferred: In body, Out body, fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Fuse Mediation Router will exclude any non-serializable objects and log it at WARN level.
disconnectOnNoReply	true	If sync is enabled then this option dictates NettyConsumer if it should disconnect where there is no reply to send back.
noReplyLogLevel	WARN	If sync is enabled this option dictates NettyConsumer which logging level to use when logging a there is no reply to send back. Values are: `FATAL, ERROR, INFO, DEBUG, OFF`.
allowDefaultCodec	true	**Camel 2.4:** The netty component installs a default codec if both, encoder/deocder is null and textline is false. Setting allowDefaultCodec to false prevents the netty component from installing a default codec as the first element in the filter chain.
textline	false	**Camel 2.4:** Only used for TCP. If no codec is specified, you can use this flag to indicate a text line based codec; if not specified or the value is false, then Object Serialization is assumed over TCP.
delimiter	LINE	**Camel 2.4:** The delimiter to use for the textline codec. Possible values are `LINE` and `NULL`.
decoderMaxLineLength	1024	**Camel 2.4:** The max line length to use for the textline codec.
autoAppendDelimiter	true	**Camel 2.4:** Whether or not to auto append missing end delimiter when sending using the textline codec.
encoding	null	**Camel 2.4:** The encoding (a charset name) to use for the textline codec. If not provided, Camel will use the JVM default Charset.

Registry based Options

Codec Handlers and SSL Keystores can be enlisted in the Registry, such as in the Spring XML file. The values that could be passed in, are the following:

Name	Description
`passphrase`	password setting to use in order to encrypt/decrypt payloads sent using SSH
`keyStoreFormat`	keystore format to be used for payload encryption. Defaults to "JKS" if not set
`securityProvider`	Security provider to be used for payload encryption. Defaults to "SunX509" if not set.
`keyStoreFile`	Client side certificate keystore to be used for encryption
`trustStoreFile`	Server side certificate keystore to be used for encryption
`sslHandler`	Reference to a class that could be used to return an SSL Handler
`encoder`	A custom Handler class that can be used to perform special marshalling of outbound payloads. Must override `org.jboss.netty.channel.ChannelDownStreamHandler`.
`encorders`	A list of encoder to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with # so Fuse Mediation Router knows it should lookup.
`decoder`	A custom Handler class that can be used to perform special marshalling of inbound payloads. Must override `org.jboss.netty.channel.ChannelUpStreamHandler`.
`decoders`	A list of decorder to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with # so Fuse Mediation Router knows it should lookup.

Netty Producer

In Producer mode, the component provides the ability to send payloads to a socket endpoint using either TCP or UDP protocols (with optional SSL support).

The producer mode supports both one-way and request-response based operations.

Netty Consumer

In Consumer mode, the component provides the ability to:

- listen on a specified socket using either TCP or UDP protocols (with optional SSL support),
- receive requests on the socket using text/xml, binary and serialized object based payloads and

- send them along on a route as message exchanges.

The consumer mode supports both one-way and request-response based operations.

A UDP Netty endpoint using Request-Reply and serialized object payload

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
    from("netty:udp://localhost:5155?sync=true")
      .process(new Processor() {
         public void process(Exchange exchange) throws Exception {
           Poetry poetry = (Poetry) exchange.getIn().getBody();
           poetry.setPoet("Dr. Sarojini Naidu");
           exchange.getOut().setBody(poetry);
         }
       }
    }
};
```

A TCP based Netty consumer endpoint using One-way communication

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
       from("netty:tcp://localhost:5150")
           .to("mock:result");
  }
};
```

An SSL/TCP based Netty consumer endpoint using Request-Reply communication

```
JndiRegistry registry = new JndiRegistry(createJndiContext());
registry.bind("password", "changeit");
registry.bind("ksf", new File("src/test/resources/keystore.jks"));
registry.bind("tsf", new File("src/test/resources/keystore.jks"));

context.createRegistry(registry);
context.addRoutes(new RouteBuilder() {
  public void configure() {
      String netty_ssl_endpoint =
         "netty:tcp://localhost:5150?sync=true&ssl=true&passphrase=#password"
         + "&keyStoreFile=#ksf&trustStoreFile=#tsf";
      String return_string =
         "When You Go Home, Tell Them Of Us And Say,"
         + "For Your Tomorrow, We Gave Our Today.";
```

```
      from(netty_ssl_endpoint)
       .process(new Processor() {
          public void process(Exchange exchange) throws Exception {
            exchange.getOut().setBody(return_string);
          }
       }
  }
});
```

Using Multiple Codecs

In certain cases it may be necessary to add chains of encoders and decoders to the netty pipeline. To add multpile codecs to a Fuse Mediation Router netty endpoint the 'encoders' and 'decoders' uri parameters should be used. Like the 'encoder' and 'decoder' parameters they are used to supply references (to lists of ChannelUpstreamHandlers and ChannelDownstreamHandlers) that should be added to the pipeline. Note that if encoders is specified then the encoder param will be ignored, similarly for decoders and the decoder param.

The lists of codecs need to be added to the Fuse Mediation Router's registry so they can be resolved when the endpoint is created.

```
LengthFieldBasedFrameDecoder lengthDecoder = new LengthFieldBasedFrameDecoder(1048576, 0,
4, 0, 4);
StringDecoder stringDecoder = new StringDecoder();
registry.bind("length-decoder", lengthDecoder);
registry.bind("string-decoder", stringDecoder);

LengthFieldPrepender lengthEncoder = new LengthFieldPrepender(4);
StringEncoder stringEncoder = new StringEncoder();
registry.bind("length-encoder", lengthEncoder);
registry.bind("string-encoder", stringEncoder);

List<ChannelUpstreamHandler> decoders = new ArrayList<ChannelUpstreamHandler>();
decoders.add(lengthDecoder);
decoders.add(stringDecoder);

List<ChannelDownstreamHandler> encoders = new ArrayList<ChannelDownstreamHandler>();
encoders.add(lengthEncoder);
encoders.add(stringEncoder);

registry.bind("encoders", encoders);
registry.bind("decoders", decoders);
```

Spring's native collections support can be used to specify the codec lists in an application context

```
    <util:list id="decoders" list-class="java.util.LinkedList">
        <bean class="org.jboss.netty.handler.codec.frame.LengthFieldBasedFrameDecoder">
            <constructor-arg value="1048576"/>
```

```
            <constructor-arg value="0"/>
            <constructor-arg value="4"/>
            <constructor-arg value="0"/>
            <constructor-arg value="4"/>
        </bean>
        <bean class="org.jboss.netty.handler.codec.string.StringDecoder"/>
    </util:list>

    <util:list id="encoders" list-class="java.util.LinkedList">
        <bean class="org.jboss.netty.handler.codec.frame.LengthFieldPrepender">
            <constructor-arg value="4"/>
        </bean>
        <bean class="org.jboss.netty.handler.codec.string.StringEncoder"/>
    </util:list>

   <bean id="length-encoder" class="org.jboss.netty.handler.codec.frame.LengthFieldPrepend
er">
        <constructor-arg value="4"/>
    </bean>
    <bean id="string-encoder" class="org.jboss.netty.handler.codec.string.StringEncoder"/>


    <bean id="length-decoder" class="org.jboss.netty.handler.codec.frame.LengthFieldBased
FrameDecoder">
        <constructor-arg value="1048576"/>
        <constructor-arg value="0"/>
        <constructor-arg value="4"/>
        <constructor-arg value="0"/>
        <constructor-arg value="4"/>
    </bean>
    <bean id="string-decoder" class="org.jboss.netty.handler.codec.string.StringDecoder"/>

</beans>
```

The bean names can then be used in netty endpoint definitions either as a comma separated list or contained in a List e.g.

```
                from("direct:multiple-codec").to("netty:tcp://localhost:5150?encoders=#en
coders&sync=false");

                from("netty:tcp://localhost:5150?decoders=#length-decoder,#string-de
coder&sync=false").to("mock:multiple-codec");
            }
        };
    }
}
```

or via spring.

```
<camelContext id="multiple-netty-codecs-context" xmlns="ht
tp://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:multiple-codec"/>
        <to uri="netty:tcp://localhost:5150?encoders=#encoders&ync=false"/>
    </route>
    <route>
        <from uri="netty:tcp://localhost:5150?decoders=#length-decoder,#string-de
coder&ync=false"/>
        <to uri="mock:multiple-codec"/>
    </route>
</camelContext>
```

Closing Channel When Complete

When acting as a server you sometimes want to close the channel when, for example, a client conversion is finished. You can do this by simply setting the endpoint option `disconnect=true`.

However you can also instruct Fuse Mediation Router on a per message basis as follows. To instruct Fuse Mediation Router to close the channel, you should add a header with the key `CamelNettyCloseChannelWhenComplete` set to a boolean `true` value. For instance, the example below will close the channel after it has written the bye message back to the client:

```
        from("netty:tcp://localhost:8080").process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                String body = exchange.getIn().getBody(String.class);
                exchange.getOut().setBody("Bye " + body);
                // some condition which determines if we should close
                if (close) {
                   exchange.getOut().setHeader(NettyConstants.NETTY_CLOSE_CHANNEL_WHEN_COM
PLETE, true);
                }
            }
        });
```

Adding custom channel pipeline factories to gain complete control over a created pipeline

Available as of Camel 2.5

Custom channel pipelines provide complete control to the user over the handler/interceptor chain by inserting custom handler(s), encoder(s) and decoders without having to specify them in the Netty Endpoint URL in a very simple way.

In order to add a custom pipeline, a custom channel pipeline factory must be created and registered with the context through the context registry (`JNDIRegistry`,or the Spring `ApplicationContextRegistry` etc).

A custom pipeline factory must be constructed as follows

- A Producer linked channel pipeline factory must extend the abstract class, `ClientPipelineFactory`.
- A Consumer linked channel pipeline factory must extend the abstract class, `ServerPipelineFactory`.
- The classes can optionally override the `getPipeline()` method in order to insert custom handler(s), encoder(s) and decoder(s). Not overriding the `getPipeline()` method creates a pipeline with no handlers, encoders or decoders wired to the pipeline.

The example below shows how ServerChannel Pipeline factory may be created

```
public class SampleServerChannelPipelineFactory extends ServerPipelineFactory {
    private int maxLineSize = 1024;
    private boolean invoked;

    public ChannelPipeline getPipeline() throws Exception {
        invoked = true;

        ChannelPipeline channelPipeline = Channels.pipeline();

        channelPipeline.addLast("encoder-SD", new StringEncoder(CharsetUtil.UTF_8));
       channelPipeline.addLast("decoder-DELIM", new DelimiterBasedFrameDecoder(maxLineSize,
 true, Delimiters.lineDelimiter()));
        channelPipeline.addLast("decoder-SD", new StringDecoder(CharsetUtil.UTF_8));
        channelPipeline.addLast("handler", new ServerChannelHandler(consumer));

        return channelPipeline;
    }

    public boolean isfactoryInvoked() {
        return invoked;
    }
}
```

The custom channel pipeline factory can then be added to the registry and instantiated/utilized on a camel route as follows:

```
Registry registry = camelContext.getRegistry();
serverPipelineFactory = new TestServerChannelPipelineFactory();
registry.bind("spf", serverPipelineFactory);
context.addRoutes(new RouteBuilder() {
    public void configure() {
        String netty_ssl_endpoint =
          "netty:tcp://localhost:5150?serverPipelineFactory=#spf";
```

```
        String return_string =
          "When You Go Home, Tell Them Of Us And Say,"
          + "For Your Tomorrow, We Gave Our Today.";

        from(netty_ssl_endpoint)
          .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
              exchange.getOut().setBody(return_string);
            }
          }
    }
});
```

Chapter 60. NMR

NMR Component

The **nmr** component is an adapter to the Normalized Message Router (NMR) in ServiceMix[1], which is intended for use by Fuse Mediation Router applications deployed directly into the OSGi container. By contrast, the JBI on page 259 component is intended for use by Fuse Mediation Router applications deployed into the ServiceMix JBI container.

Installing

The NMR component is provided with Apache ServiceMix. It is **not** distributed with Fuse Mediation Router. To install the NMR component in ServiceMix, enter the following command in the ServiceMix console window:

```
features install nmr
```

You also need to instantiate the NMR component. You can do this by editing your Spring configuration file, `META-INF/spring/*.xml`, and adding the following `bean` instance:

```
<beans xmlns:osgi="http://www.springframework.org/schema/osgi" ... >
    ...
    <bean id="nmr" class="org.apache.servicemix.camel.nmr.ServiceMixComponent">
        <property name="nmr">
            <osgi:reference interface="org.apache.servicemix.nmr.api.NMR" />
        </property>
    </bean>
    ...
</beans>
```

NMR consumer and producer endpoints

The following code:

```
from("nmr:MyServiceEndpoint")
```

Automatically exposes a new endpoint to the bus with endpoint name `MyServiceEndpoint` (see #URI-format).

When an NMR endpoint appears at the end of a route, for example:

```
to("nmr:MyServiceEndpoint")
```

The messages sent by this producer endpoint are sent to the already deployed JBI endpoint.

[1] http://servicemix.apache.org/home.html

URI format

```
nmr:endpointName
```

URI Options

Option	Default Value	Description
`synchronous`	`false`	When this is set to **`true`** on a consumer endpoint, an incoming, synchronous NMR Exchange will be handled on the sender's thread instead of being handled on a new thread of the NMR endpoint's thread pool

Examples

```
from("nmr:MyServiceEndpoint")
from("nmr:MyServiceEndpoint?synchronous=true").to("nmr:AnotherEndpoint")
```

Using Stream bodies

If you are using a stream type as the message body, you should be aware that a stream is only capable of being read once. So if you enable `DEBUG` logging, the body is usually logged and thus read. To deal with this, Camel has a `streamCaching` option that can cache the stream, enabling you to read it multiple times.

```
from("nmr:MyEndpoint").streamCaching().to("xslt:transform.xsl", "bean:doSomething");
```

From **Camel 1.5** onwards, the stream caching is default enabled, so it is not necessary to set the `streamCaching()` option. In **Camel 2.0** we store big input streams (by default, over 64K) in a `temp` file using `CachedOutputStream`. When you close the input stream, the temp file will be deleted.

Chapter 61. Pax-Logging

PaxLogging component

Available in Camel 2.6

The `paxlogging` component can be used in an OSGi environment to receive PaxLogging[1] events and process them.

Dependencies

Maven users need to add the following dependency to their `pom.xml`

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-paxlogging</artifactId>
  <version>${camel-version}</version>
</dependency>
```

where `$\{camel-version\}` must be replaced by the actual version of Camel (2.6.0 or higher).

URI format

```
paxlogging:appender
```

where `appender` is the name of the pax appender that need to be configured in the PaxLogging service configuration.

URI options

Name	Default value	Description

Message headers

Name	Type	Message	Description

[1] http://wiki.ops4j.org/display/paxlogging/Pax+Logging

Message body

The `in` message body will be set to the received PaxLoggingEvent.

Example usage

```
<route>
    <from uri="paxlogging:camel"/>
    <to uri="stream:out"/>
</route>
```

Configuration:

```
log4j.rootLogger=INFO, out, osgi:VmLogAppender, osgi:camel
```

Chapter 62. Pojo

Pojo Component

The **pojo:** component is now just an alias for the Bean on page 41 component.

Has been removed in Fuse Mediation Router 2.0.

Chapter 63. Printer

Printer Component

Available as of Fuse Mediation Router 2.1

The **printer** component provides a way to direct payloads on a route to a printer. Obviously the payload has to be a formatted piece of payload in order for the component to appropriately print it. The objective is to be able to direct specific payloads as jobs to a line printer in a Fuse Mediation Router flow.

This component only supports a producer endpoint.

The functionality allows for the payload to be printed on a default printer, named local, remote or wirelessly linked printer using the javax printing API under the covers.

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

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

URI format

Since the URI scheme for a printer has not been standardized (the nearest thing to a standard being the IETF print standard) and therefore not uniformly applied by vendors, we have chosen **"lpr"** as the scheme.

```
lpr://localhost/default[?options]
lpr://remotehost:port/path/to/printer[?options]
```

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

Options

Name	Default Value	Description

`mediaSize`	`MediaSizeName.NA_LETTER`	Sets the stationary as defined by enumeration settings in the javax.print.attribute.standard.MediaSizeName API[1] API. The default setting is to use North American Letter sized stationary
`copies`	`1`	Sets number of copies based on the `javax.print.attribute.standard.Copies` API
`sides`	`Sides.ONE_SIDED`	Sets one sided or two sided printing based on the `javax.print.attribute.standard.Sides` API
`flavor`	`DocFlavor.BYTE_ARRAY`	Sets `DocFlavor` based on the `javax.print.DocFlavor` API
`mimeType`	`AUTOSENSE`	Sets `mimeTypes` supported by the `javax.print.DocFlavor` API

Printer Producer

Sending data to the printer is very straightforward and involves creating a producer endpoint that can be sent message exchanges on in route.

Example 1: Printing text based payloads on a Default printer using letter stationary and one-sided mode

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
       from(file://inputdir/?delete=true)
       .to("lpr://localhost/default?copies=2" +
           "&flavor=DocFlavor.INPUT_STREAM&" +
           "&mimeType=AUTOSENSE" +
           "&mediaSize=na-letter" +
           "&sides=one-sided")
    }};
```

Example 2: Printing GIF based payloads on a Remote printer using A4 stationary and one-sided mode

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
       from(file://inputdir/?delete=true)
       .to("lpr://remotehost/sales/salesprinter" +
           "?copies=2&sides=one-sided" +
           "&mimeType=GIF&mediaSize=iso-a4" +
           "&flavor=DocFlavor.INPUT_STREAM")
   }};
```

[1] http://download.oracle.com/javase/6/docs/api/javax/print/attribute/standard/MediaSizeName.html

Example 3: Printing JPEG based payloads on a Remote printer using Japanese Postcard stationary and one-sided mode

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
       from(file://inputdir/?delete=true)
       .to("lpr://remotehost/sales/salesprinter" +
           "?copies=2&sides=one-sided" +
           "&mimeType=JPEG" +
           "&mediaSize=japanese-postcard" +
           "&flavor=DocFlavor.INPUT_STREAM")
    }};
```

Chapter 64. Properties

Properties Component

Available as of Fuse Mediation Router 2.3

URI format

```
properties:key[?options]
```

Where **key** is the key for the property to lookup

Options

Name	Type	Default	Description
`cache`	`boolean`	`true`	Whether or not to cache loaded properties.
`locations`	`String`	`null`	A list of locations to load properties. You can use comma to separate multiple locations. This option will override any default locations and **only** use the locations from this option.

See also

- Jasypt on page 253 for using encrypted values (for example, passwords) in the properties

Chapter 65. Quartz

Quartz Component

The **quartz:** component provides a scheduled delivery of messages using the Quartz scheduler[1]. Each endpoint represents a different timer (in Quartz terms, a Trigger and JobDetail).

URI format

```
quartz://timerName?options
quartz://groupName/timerName?options
quartz://groupName/timerName/cronExpression       (@deprecated)
quartz://groupName/timerName/?cron=expression     (Fuse Mediation Router 2.0)
quartz://timerName?cron=expression                (Fuse Mediation Router 2.0)
```

The component uses either a `CronTrigger` or a `SimpleTrigger`. If no cron expression is provided, the component uses a simple trigger. If no `groupName` is provided, the quartz component uses the `Camel` group name.

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

Options

Parameter	Default	Description
`cron`	*None*	Specifies a `cron` expression (not compatible with the `trigger.\*` or `job.\*` options).
`trigger.repeatCount`	`0`	SimpleTrigger: How many times should the timer repeat?
`trigger.repeatInterval`	`0`	SimpleTrigger: The amount of time in milliseconds between repeated triggers.
`job.name`	`null`	Sets the job name.
`job._XXX_`	`null`	Sets the job option with the `_XXX_` setter name.
`trigger._XXX_`	`null`	Sets the trigger option with the `_XXX_` setter name.
`stateful`	`false`	Uses a Quartz `StatefulJob` instead of the default job.
`fireNow`	`false`	New to Camel 2.2.0, if it is true will fire the trigger when the route is start when using SimpleTrigger.

For example, the following routing rule will fire two timer events to the `mock:results` endpoint:

[1] http://www.opensymphony.com/quartz/

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

When using a StatefulJob[2], the JobDataMap[3] is re-persisted after every execution of the job, thus preserving state for the next execution.

Configuring quartz.properties file

By default Quartz will look for a `quartz.properties` file in the root of the classpath. If you are using WAR deployments this means just drop the quartz.properties in `WEB-INF/classes`.

However the Camel Quartz on page 389 component also allows you to configure properties:

Parameter	Default	Type	Description
`properties`	`null`	`Properties`	**Camel 2.4**: You can configure a `java.util.Propoperties` instance.
`propertiesFile`	`null`	`String`	**Camel 2.4**: File name of the properties to load from the classpath

To do this you can configure this in Spring XML as follows

```
<bean id="quartz" class="org.apache.camel.component.quartz.QuartzComponent">
    <property name="propertiesFile" value="com/mycompany/myquartz.properties"/>
</bean>
```

Starting the Quartz scheduler

Available as of Camel 2.4

The Quartz on page 389 component offers an option to let the Quartz scheduler be started delayed, or not auto started at all.

Parameter	Default	Type	Description
`startDelayedSeconds`	`0`	`int`	**Camel 2.4**: Seconds to wait before starting the quartz scheduler.
`autoStartScheduler`	`true`	`boolean`	**Camel 2.4:** Whether or not the scheduler should be auto started.

To do this you can configure this in Spring XML as follows

```
<bean id="quartz" class="org.apache.camel.component.quartz.QuartzComponent">
    <property name="startDelayedSeconds" value="5"/>
</bean>
```

[2] http://www.quartz-scheduler.org/docs/api/org/quartz/StatefulJob.html
[3] http://www.quartz-scheduler.org/docs/api/org/quartz/JobDataMap.html

Clustering

Available as of Camel 2.4

If you use Quartz in clustered mode, e.g. the `JobStore` is clustered. Then from Camel 2.4 onwards the Quartz on page 389 component will **not** pause/remove triggers when a node is being stopped/shutdown. This allows the trigger to keep running on the other nodes in the cluster.

Note

When running in clustered node, no checking is done to ensure unique job name/group for endpoints.

Message Headers

Fuse Mediation Router adds the getters from the Quartz Execution Context as header values. The following headers are added: `calendar`, `fireTime`, `jobDetail`, `jobInstance`, `jobRuntTime`, `mergedJobDataMap`, `nextFireTime`, `previousFireTime`, `refireCount`, `result`, `scheduledFireTime`, `scheduler`, `trigger`, `triggerName`, `triggerGroup`.

The `fireTime` header contains the `java.util.Date` of when the exchange was fired.

Using Cron Triggers

Avaiable as of Fuse Mediation Router 2.0 Quartz supports Cron-like expressions[4] for specifying timers in a handy format. You can use these expressions in the `cron` URI parameter; though to preserve valid URI encoding we allow + to be used instead of spaces. Quartz provides a little tutorial[5] on how to use cron expressions.

For example the following will fire a message every five minutes starting at 12pm (noon) to 6pm on weekdays:

```
from("quartz://myGroup/myTimerName?cron=0+0/5+12-18+?+*+MON-FRI").to("act
ivemq:Totally.Rocks");
```

which is equivalent to using the cron expression

```
0 0/5 12-18 ? * MON-FRI
```

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

[4] http://www.opensymphony.com/quartz/api/org/quartz/CronTrigger.html
[5] http://www.opensymphony.com/quartz/wikidocs/CronTriggers%20Tutorial.html

URI Character	Cron character
`\+`	*Space*

Using Cron Triggers in Fuse Mediation Router 1.x

@deprecated Quartz supports Cron-like expressions[6] 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 endpoint URI 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
`/`	*Space*
`$`	?

See also:

- Timer on page 507

[6] http://www.opensymphony.com/quartz/api/org/quartz/CronTrigger.html

Chapter 66. Queue

Queue Component

Deprecated

To avoid confusion with JMS queues, this component is now deprecated in 1.1 onwards. Please use the SEDA on page 429 component instead

The **queue:** component provides asynchronous SEDA[1] behaviour so that messages are exchanged on a BlockingQueue[2] and consumers are invoked in a seperate thread pool to the producer.

Note that queues are only visible within a single CamelContext. If you want to communicate across CamelContext instances such as to communicate across web applications, see the VM on page 519 component.

Note also that this component has nothing to do with JMS on page 293, if you want a distributed SEA then try using either JMS on page 293 or ActiveMQ on page 25 or even MINA on page 349

URI format

```
queue:someName
```

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

[1] http://www.eecs.harvard.edu/~mdw/proj/seda/
[2] http://java.sun.com/j2se/1.5.0/docs/api/java/util/BlockingQueue.html

Chapter 67. Quickfix

QuickFIX/J Component

Available as of Camel 2.0

The **quickfix** component adapts the QuickFIX/J[1] FIX engine for using in Camel . This component uses the standard Financial Interchange (FIX) protocol[2] for message transport.

Previous Versions

The **quickfix** component was rewritten for Camel 2.5. For information about using the **quickfix** component prior to 2.5 see the documentation section below.

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

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

URI format

```
quickfix:configFile[?sessionID=sessionID]
```

The **configFile** is the name of the QuickFIX/J configuration to use for the FIX engine (located as a resource found in your classpath). The optional sessionID identifies a specific FIX session. The format of the sessionID is:

```
(BeginString):(SenderCompID)[/(SenderSubID)[/(SenderLocationID)]]->(TargetCompID)[/(Target
SubID)[/(TargetLocationID)]]
```

Example URIs:

```
quickfix:config.cfg

quickfix:config.cfg?sessionID=FIX.4.2:MyTradingCompany->SomeExchange
```

[1] http://www.quickfixj.org/
[2] http://www.fixprotocol.org/

Endpoints

FIX sessions are endpoints for the **quickfix** component. An endpoint URI may specify a single session or all sessions managed by a specific QuickFIX/J engine. Typical applications will use only one FIX engine but advanced users may create multiple FIX engines by referencing different configuration files in **quickfix** component endpoint URIs.

When a consumer does not include a session ID in the endpoint URI, it will receive exchanges for all sessions managed by the FIX engine associated with the configuration file specified in the URI. If a producer does not specify a session in the endpoint URI then it must include the session-related fields in the FIX message being sent. If a session is specified in the URI then the component will automatically inject the session-related fields into the FIX message.

Exchange Format

The exchange headers include information to help with exchange filtering, routing and other processing. The following headers are available:

Header Name	Description
`EventCategory`	One of `AppMessageReceived`, `AppMessageSent`, `AdminMessageReceived`, `AdminMessageSent`, `SessionCreated`, `SessionLogon`, `SessionLogoff`. See the `QuickfixjEventCategory` enum.
`SessionID`	The FIX message SessionID
`MessageType`	The FIX MsgType tag value
`DataDictionary`	Specifies a data dictionary to used for parsing an incoming message. Can be an instance of a data dictionary or a resource path for a QuickFIX/J data dictionary file

The DataDictionary header is useful if string messages are being received and need to be parsed in a route. QuickFIX/J requires a data dictionary to parse certain types of messages (with repeating groups, for example). By injecting a DataDictionary header in the route after receiving a message string, the FIX engine can properly parse the data.

QuickFIX/J Configuration Extensions

When using QuickFIX/J directly, one typically writes code to create instances of logging adapters, message stores and communication connectors. The **quickfix** component will automatically create instances of these classes based on information in the configuration file. It also provides defaults for many of the common required settings and adds additional capabilities (like the ability to activate JMX support).

The following sections describe how the **quickfix** component processes the QuickFIX/J configuration. For comprehensive information about QuickFIX/J configuration, see the QFJ user manual[3].

[3] http://www.quickfixj.org/quickfixj/usermanual/usage/configuration.html

Communication Connectors

When the component detects an initiator or acceptor session setting in the QuickFIX/J configuration file it will automatically create the corresponding initiator and/or acceptor connector. These settings can be in the default or in a specific session section of the configuration file.

Session Setting	Component Action
`ConnectionType=initiator`	Create an initiator connector
`ConnectionType=acceptor`	Create an acceptor connector

The threading model for the QuickFIX/J session connectors can also be specified. These settings affect all sessions in the configuration file and must be placed in the settings default section.

Default/Global Setting	Component Action
`ThreadModel=ThreadPerConnector`	Use `SocketInitiator` or `SocketAcceptor` (default)
`ThreadModel=ThreadPerSession`	Use `ThreadedSocketInitiator` or `ThreadedSocketAcceptor`

Logging

The QuickFIX/J logger implementation can be specified by including the following settings in the default section of the configuration file. The `ScreenLog` is the default if none of the following settings are present in the configuration. It's an error to include settings that imply more than one log implementation.

Default/Global Setting	Component Action
`ScreenLogShowEvents`	Use a `ScreenLog`
`ScreenLogShowIncoming`	Use a `ScreenLog`
`ScreenLogShowOutgoing`	Use a `ScreenLog`
`SLF4J*`	**Camel 2.6+**. Use a `SLF4JLog`. Any of the SLF4J settings will cause this log to be used.
`FileLogPath`	Use a `FileLog`
`JdbcDriver`	Use a `JdbcLog`

Message Store

The QuickFIX/J message store implementation can be specified by including the following settings in the default section of the configuration file. The `MemoryStore` is the default if none of the following settings are

present in the configuration. It's an error to include settings that imply more than one message store implementation.

Default/Global Setting	Component Action
`JdbcDriver`	Use a `JdbcStore`
`FileStorePath`	Use a `FileStore`
`SleepycatDatabaseDir`	Use a `SleepcatStore`

Message Factory

A message factory is used to construct domain objects from raw FIX messages. The default message factory is `DefaultMessageFactory`. However, advanced applications may require a custom message factory. This can be set on the QuickFIX/J component.

JMX

Default/Global Setting	Component Action
`UseJmx`	if `Y`, then enable QuickFIX/J JMX

Other Defaults

The component provides some default settings for what are normally required settings in QuickFIX/J configuration files. `SessionStartTime` and `SessionEndTime` default to "00:00:00", meaning the session will not be automatically started and stopped. The `HeartBtInt` (heartbeat interval) defaults to 30 seconds.

Minimal Initiator Configuration Example

```
[SESSION]
ConnectionType=initiator
BeginString=FIX.4.4
SenderCompID=YOUR_SENDER
TargetCompID=YOUR_TARGET
```

Spring Configuration

Camel 2.6+

The QuickFIX/J component includes a Spring `FactoryBean` for configuring the session settings within a Spring context. A type converter for QuickFIX/J session ID strings is also included. The following example shows a simple configuration of an acceptor and initiator session with default settings for both sessions.

```
<!-- camel route -->
      <camelContext id="quickfixjContext" xmlns="http://camel.apache.org/schema/spring">
      <route>
      <from uri="quickfix:example"/>
      <filter>
      <simple>${in.header.EventCategory} == 'AppMessageReceived'</simple>
      <to uri="log:test"/>
      </filter>
      </route>
      </camelContext>

      <!-- quickfix component -->
      <bean id="quickfix" class="org.apache.camel.component.quickfixj.QuickfixjComponent">

      <property name="engineSettings">
      <util:map>
      <entry key="quickfix:example" value-ref="quickfixjSettings"/>
      </util:map>
      </property>
      <property name="messageFactory">
      <bean class="org.apache.camel.component.quickfixj.QuickfixjSpringTest.CustomMessage
Factory"/>
      </property>
      </bean>

      <!-- quickfix settings -->
      <bean id="quickfixjSettings"
      class="org.apache.camel.component.quickfixj.QuickfixjSettingsFactory">
      <property name="defaultSettings">
      <util:map>
      <entry key="SocketConnectProtocol" value="VM_PIPE"/>
      <entry key="SocketAcceptProtocol" value="VM_PIPE"/>
      <entry key="UseDataDictionary" value="N"/>
      </util:map>
      </property>
      <property name="sessionSettings">
      <util:map>
      <entry key="FIX.4.2:INITIATOR->ACCEPTOR">
      <util:map>
      <entry key="ConnectionType" value="initiator"/>
      <entry key="SocketConnectHost" value="localhost"/>
      <entry key="SocketConnectPort" value="5000"/>
      </util:map>
      </entry>
      <entry key="FIX.4.2:ACCEPTOR->INITIATOR">
      <util:map>
      <entry key="ConnectionType" value="acceptor"/>
      <entry key="SocketAcceptPort" value="5000"/>
      </util:map>
```

```
    </entry>
    </util:map>
    </property>
    </bean>
```

Exception handling

QuickFIX/J behavior can be modified if certain exceptions are thrown during processing of a message. If a `RejectLogon` exception is thrown while processing an incoming logon administrative message, then the logon will be rejected.

Normally, QuickFIX/J handles the logon process automatically. However, sometimes an outgoing logon message must be modified to include credentials required by a FIX counterparty. If the FIX logon message body is modified when sending a logon message (EventCategory={{AdminMessageSent}} the modified message will be sent to the counterparty. It is important that the outgoing logon message is being processed *synchronously*. If it is processed asynchronously (on another thread), the FIX engine will immediately send the unmodified outgoing message when it's callback method returns.

FIX Sequence Number Management

If an application exception is thrown during *synchronous* exchange processing, this will cause QuickFIX/J to not increment incoming FIX message sequence numbers and will cause a resend of the counterparty message. This FIX protocol behavior is primarily intended to handle *transport* errors rather than application errors. There are risks associated with using this mechanism to handle application errors. The primary risk is that the message will repeatedly cause application errors each time it's re-received. A better solution is to persist the incoming message (database, JMS queue) immediately before processing it. This also allows the application to process messages asynchronously without losing messages when errors occur.

Although it's possible to send messages to a FIX session before it's logged on (the messages will be sent at logon time), it is usually a better practice to wait until the session is logged on. This eliminates the required sequence number resynchronization steps at logon. Waiting for session logon can be done by setting up a route that processes the `SessionLogon` event category and signals the application to start sending messages.

See the FIX protocol specifications and the QuickFIX/J documentation for more details about FIX sequence number management.

Route Examples

Several examples are included in the QuickFIX/J component source code (test subdirectories). One of these examples implements a trival trade excecution simulation. The example defines an application component that uses the URI scheme "trade-executor".

The following route receives messages for the trade executor session and passes application messages to the trade executor component.

```
from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:MARKET->TRADER").
    filter(header(QuickfixjEndpoint.EVENT_CATEGORY_KEY).isEqualTo(QuickfixjEventCat
egory.AppMessageReceived)).
    to("trade-executor:market");
```

The trade executor component generates messages that are routed back to the trade session. The session ID must be set in the FIX message itself since no session ID is specified in the endpoint URI.

```
from("trade-executor:market").to("quickfix:examples/inprocess.cfg");
```

The trader session consumes execution report messages from the market and processes them.

```
from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:TRADER->MARKET").
   filter(header(QuickfixjEndpoint.MESSAGE_TYPE_KEY).isEqualTo(MsgType.EXECUTION_REPORT)).

    bean(new MyTradeExecutionProcessor());
```

QuickFIX/J Component Prior to Camel 2.5

Available since Camel 2.0

The **quickfix** component is an implementation of the QuickFIX/J[4] engine for Java . This engine allows to connect to a FIX server which is used to exchange financial messages according to FIX protocol[5] standard.

Note: The component can be used to send/receives messages to a FIX server.

URI format

```
quickfix-server:config file
quickfix-client:config file
```

Where **config file** is the location (in your classpath) of the quickfix configuration file used to configure the engine at the startup.

Note: Information about parameters available for quickfix can be found on QuickFIX/J[6] web site.

The quickfix-server endpoint must be used to receive from FIX server FIX messages and quickfix-client endpoint in the case that you want to send messages to a FIX gateway.

[4] http://www.quickfixj.org/
[5] http://www.fixprotocol.org/
[6] http://www.quickfixj.org/quickfixj/usermanual/usage/configuration.html

Exchange data format

The QuickFIX/J engine is like CXF component a messaging bus using MINA as protocol layer to create the socket connection with the FIX engine gateway.

When QuickFIX/J engine receives a message, then it create a QuickFix.Message instance which is next received by the camel endpoint. This object is a 'mapping object' created from a FIX message formatted initially as a collection of key value pairs data. You can use this object or you can use the method 'toString' to retrieve the original FIX message.

Note: Alternatively, you can use camel bindy dataformat[7] to transform the FIX message into your own java POJO

When a message must be send to QuickFix, then you must create a QuickFix.Message instance.

Samples

Direction : to FIX gateway

```
<route>
  <from uri="activemq:queue:fix"/>
  <bean ref="fixService" method="createFixMessage" /> // bean method in charge to transform
 message into a QuickFix.Message
  <to uri="quickfix-client:META-INF/quickfix/client.cfg" /> // Quickfix engine who will
send the FIX messages to the gateway
</route>
```

Direction : from FIX gateway

```
<route>
  <from uri="quickfix-server:META-INF/quickfix/server.cfg"/> // QuickFix engine who will
receive the message from FIX gateway
  <bean ref="fixService" method="parseFixMessage" /> // bean method parsing the QuickFix.Mes
sage
  <to uri="uri="activemq:queue:fix"/>" />
</route>
```

[7] bindy

Chapter 68. Ref

Ref Component

The **ref:** component is used for lookup of existing endpoints bound in the Registry.

URI format

```
ref:someName
```

Where **someName** is the name of an endpoint in the Registry (usually, but not always, the Spring registry). If you are using the Spring registry, `someName` would be the bean ID of an endpoint in the Spring registry.

Runtime lookup

This component can be used when you need dynamic discovery of endpoints in the Registry where you can compute the URI at runtime. Then you can look up the endpoint using the following code:

```
   // lookup the endpoint
   String myEndpointRef = "bigspenderOrder";
   Endpoint endpoint = context.getEndpoint("ref:" + myEndpointRef);

   Producer producer = endpoint.createProducer();
   Exchange exchange = producer.createExchange();
   exchange.getIn().setBody(payloadToSend);
   // send the exchange
   producer.process(exchange);
   ...
```

And you could have a list of endpoints defined in the Registry such as:

```
  <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
      <endpoint id="normalOrder" uri="activemq:order.slow"/>
      <endpoint id="bigspenderOrder" uri="activemq:order.high"/>
      ...
  </camelContext>
```

Sample

In the sample below we use the `ref:` in the URI to reference the endpoint with the spring ID, `endpoint2`:

```
<bean id="mybean" class="org.apache.camel.spring.example.DummyBean">
  <property name="endpoint" ref="endpoint1"/>
</bean>
```

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" disabled="true"/>
  <endpoint id="endpoint1" uri="direct:start"/>
  <endpoint id="endpoint2" uri="mock:end"/>

  <route>
    <from ref="endpoint1"/>
    <to uri="ref:endpoint2"/>
  </route>
</camelContext>
```

You could, of course, have used the `ref` attribute instead:

```
      <to ref="endpoint2"/>
```

Which is the more common way to write it.

Chapter 69. Restlet

Restlet Component

The **Restlet** component provides Restlet[1] based endpoints[2] for consuming and producing RESTful resources.

URI format

```
restlet:restletUrl[?options]
```

Format of restletUrl:

```
protocol://hostname[:port][/resourcePattern]
```

Restlet promotes decoupling of protocol and application concerns. The reference implementation of Restlet Engine[3] supports a number of protocols. However, we have tested the HTTP protocol only. The default port is port 80. We do not automatically switch default port based on the protocol yet.

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

Options

Name	Default Value	Description
`headerFilterStrategy=#refName` (**2.x or later**)	An instance of `RestletHeaderFilterStrategy`	Use the `#` notation (`headerFilterStrategy=#refName`) to reference a header filter strategy in the Camel Registry. The strategy will be plugged into the restlet binding if it is `HeaderFilterStrategyAware`.
`restletBindingRef` (**1.x**),	An instance of `DefaultRestletBinding`	The bean ID of a `RestletBinding` object in the Camel Registry.
`restletBinding` (**2.x or later**), `restletBinding=#`*refName*.	An instance of `DefaultRestletBinding`	The bean ID of a `RestletBinding` object in the Camel Registry.
`restletMethod`	`GET`	On a producer endpoint, specifies the request method to use. On a consumer endpoint, specifies that the endpoint consumes only `restletMethod` requests. The string

[1] http://www.restlet.org
[2] Endpoint
[3] http://www.noelios.com/products/restlet-engine

		value is converted to org.restlet.data.Method[4] by the `Method.valueOf(String)` method.
`restletMethod`	`GET`	On a producer endpoint, specifies the request method to use. On a consumer endpoint, specifies that the endpoint consumes only `restletMethod` requests. The string value is converted to org.restlet.data.Method[5] by the `Method.valueOf(String)` method.
`restletMethods` (**2.x or later**)	*None*	**Consumer only** Specify one or more methods separated by commas (e.g. `restletMethods=post,put`) to be serviced by a restlet consumer endpoint. If both `restletMethod` and `restletMethods` options are specified, the `restletMethod` setting is ignored.
`restletRealmRef` (**1.x**),	`null`	The bean ID of the Realm Map in the Camel Registry.
`restletUriPatterns=#refName` (**2.x or later**)	*None*	**Consumer only** Specify one ore more URI templates to be serviced by a restlet consumer endpoint, using the `#` notation to reference a `List<String>` in the Camel Registry. If a URI pattern has been defined in the endpoint URI, both the URI pattern defined in the endpoint and the `restletUriPatterns` option will be honored.
`throwExceptionOnFailure` (**2.6 or later**)	`true`	*Producer only* Throws exception on a producer failure.

Fuse Mediation Router 1.x Message Headers

Name	Type	Description

[4] http://www.restlet.org/documentation/1.1/api/org/restlet/data/Method.html
[5] http://www.restlet.org/documentation/1.1/api/org/restlet/data/Method.html

`org.apache.camel.restlet.auth.login`	`String`	Login name for basic authentication. It is set on the IN message by the application and gets filtered before the restlet request header by Fuse Mediation Router.
`org.apache.camel.restlet.auth.password`	`String`	Password name for basic authentication. It is set on the IN message by the application and gets filtered before the restlet request header by Fuse Mediation Router.
`org.apache.camel.restlet.mediaType`	`String`	Specifies the content type, which can be set on the OUT message by the application/processor. The value is the `content-type` of the response message. If this header is not set, the `content-type` is set based on the object type of the OUT message body.
`org.apache.camel.restlet.queryString`	`String`	The query string of the request URI. It is set on the IN message by `DefaultRestletBinding` when the restlet component receives a request.
`org.apache.camel.restlet.responseCode`	`String` or `Integer`	The response code can be set on the OUT message by the application/processor. The value is the response code of the response message. If this header is not set, the response code is set by the restlet runtime engine.
`org.restlet.*`		Attributes of a restlet message that get propagated to Fuse Mediation Router IN headers.

Fuse Mediation Router 2.0 Message Headers

Name	Type	Description
`CamelContentType`	`String`	Specifies the content type, which can be set on the OUT message by the application/processor. The value is the `content-type` of the response message. If this header is not set, the `content-type` is based on the object type of the OUT message body. In Camel 2.3 onward, if the Content-Type header is specified in the Camel IN message, the value of the header determine the content type for the Restlet request message.nbsp; Otherwise, it is defaulted to "application/x-www-form-urlencoded'. Prior to release 2.3, it is not possible to change the request content type default.
`CamelHttpMethod`	`String`	The HTTP request method. This is set in the IN message header.

`CamelHttpQuery`	`String`	The query string of the request URI. It is set on the IN message by `DefaultRestletBinding` when the restlet component receives a request.
`CamelHttpResponseCode`	`String` or `Integer`	The response code can be set on the OUT message by the application/processor. The value is the response code of the response message. If this header is not set, the response code is set by the restlet runtime engine.
`CamelHttpUri`	`String`	The HTTP request URI. This is set in the IN message header.
`CamelRestletLogin`	`String`	Login name for basic authentication. It is set on the IN message by the application and gets filtered before the restlet request header by Fuse Mediation Router.
`CamelRestletPassword`	`String`	Password name for basic authentication. It is set on the IN message by the application and gets filtered before the restlet request header by Fuse Mediation Router.
`org.restlet.*`		Attributes of a Restlet message that get propagated to Fuse Mediation Router IN headers.

Message Body

Fuse Mediation Router will store the restlet response from the external server on the OUT body. All headers from the IN message will be copied to the OUT message, so that headers are preserved during routing.

Restlet Endpoint with Authentication

The following route starts a `restlet` consumer endpoint that listens for `POST` requests on http://localhost:8080 [6]. The processor creates a response that echoes the request body and the value of the `id` header.

```
from("restlet:http://localhost:9080/securedOrders?restletMethod=post&restletRealm=#realm").pro
cess(new Processor() {
    public void process(Exchange exchange) throws Exception {
        exchange.getOut().setBody(
                "received [" + exchange.getIn().getBody()
                + "] as an order id = "
                + exchange.getIn().getHeader("id"));
    }
});
```

The `restletRealm` setting (in Fuse Mediation Router **2.x**, use the `#` notation, that is, `restletRealm=#refName`) in the URI query is used to look up a Realm Map in the registry. If this option is specified, the restlet consumer uses the information to authenticate user logins. Only *authenticated* requests can access the resources. In

[6] http://localhost:8080

this sample, we create a Spring application context that serves as a registry. The bean ID of the Realm Map should match the *restletRealmRef*.

```
<util:map id="realm">
 <entry key="admin" value="foo" />
 <entry key="bar" value="foo" />
</util:map>
```

The following sample starts a `direct` endpoint that sends requests to the server on http://localhost:8080 [7] (that is, our restlet consumer endpoint).

```
// Note: restletMethod and restletRealmRef are stripped
// from the query before a request is sent as they are
// only processed by Camel.
from("direct:start-auth").to("restlet:http://localhost:9080/securedOrders?restletMeth
od=post");
```

That is all we need. We are ready to send a request and try out the restlet component:

```
final String id = "89531";

Map<String, Object> headers = new HashMap<String, Object>();
headers.put(RestletConstants.RESTLET_LOGIN, "admin");
headers.put(RestletConstants.RESTLET_PASSWORD, "foo");
headers.put("id", id);

String response = (String) template.requestBodyAndHeaders("direct:start-auth",
        "<order foo='1'/>", headers);
```

The sample client sends a request to the `direct:start-auth` endpoint with the following headers:

- `CamelRestletLogin` (used internally by Fuse Mediation Router)
- `CamelRestletPassword` (used internally by Fuse Mediation Router)
- `id` (application header)

Note

`org.apache.camel.restlet.auth.login` and `org.apache.camel.restlet.auth.password` will not be propagated as Restlet header.

The sample client gets a response like the following:

[7] http://localhost:8080

```
received [<order foo='1'/>] as an order id = 89531
```

Single restlet endpoint to service multiple methods and URI templates (2.0 or later)

It is possible to create a single route to service multiple HTTP methods using the `restletMethods` option. This snippet also shows how to retrieve the request method from the header:

```
from("restlet:http://localhost:9080/users/{username}?restletMethods=post,get")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            // echo the method
            exchange.getOut().setBody(exchange.getIn().getHeader(Exchange.HTTP_METHOD,
                                                                 String.class));

        }
    });
```

In addition to servicing multiple methods, the next snippet shows how to create an endpoint that supports multiple URI templates using the `restletUriPatterns` option. The request URI is available in the header of the IN message as well. If a URI pattern has been defined in the endpoint URI (which is not the case in this sample), both the URI pattern defined in the endpoint and the `restletUriPatterns` option will be honored.

```
from("restlet:http://localhost:9080?restletMethods=post,get&restletUriPatterns=#uriTemplates")

    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            // echo the method
            String uri = exchange.getIn().getHeader(Exchange.HTTP_URI, String.class);
            String out = exchange.getIn().getHeader(Exchange.HTTP_METHOD, String.class);
            if ("http://localhost:9080/users/homer".equals(uri)) {
              exchange.getOut().setBody(out + " " + exchange.getIn().getHeader("username",
 String.class));
          } else if ("http://localhost:9080/atom/collection/foo/component/bar".equals(uri))
 {
               exchange.getOut().setBody(out + " " + exchange.getIn().getHeader("id",
String.class)
                                         + " " + exchange.getIn().getHeader("cid",
String.class));

            }

        }
    });
```

The `restletUriPatterns=#uriTemplates` option references the `List<String>` bean defined in the Spring XML configuration.

```
<util:list id="uriTemplates">
    <value>/users/{username}</value>
    <value>/atom/collection/{id}/component/{cid}</value>
</util:list>
```

Chapter 70. RMI

RMI Component

The **rmi:** component binds PojoExchanges[1] to the RMI protocol (JRMP).

Since this binding is just using RMI, normal RMI rules still apply regarding what methods can be invoked. This component supports only PojoExchanges[2] that carry a method invocation from an interface that extends the Remote[3] interface. All parameters in the method should be either Serializable[4] or `Remote` objects.

URI format

```
rmi://rmi-regisitry-host:rmi-registry-port/registry-path[?options]
```

For example:

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

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

Options

Name	Default Value	Description
`method`	`null`	As of **Fuse Mediation Router 1.3**, you can set the name of the method to invoke.

Using

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

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

To bind an existing camel processor or service in an RMI registry, define an RMI endpoint as follows:

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

Note that when binding an RMI consumer endpoint, you must specify the `Remote` interfaces exposed.

[1] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/pojo/PojoExchange.html
[2] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/pojo/PojoExchange.html
[3] http://java.sun.com/j2se/1.3/docs/api/java/rmi/Remote.html
[4] http://java.sun.com/j2se/1.5.0/docs/api/java/io/Serializable.html

Chapter 71. Routebox

Routebox Component

Available as of Camel 2.6

The **routebox** component enables the creation of specialized endpoints that offer encapsulation and a strategy based indirection service to a collection of camel routes hosted in an automatically created or user injected camel context.

Routebox endpoints are camel endpoints that may be invoked directly on camel routes. The routebox endpoint performs the following key functions * encapsulation - acts as a blackbox, hosting a collection of camel routes stored in an inner camel context. The inner context is fully under the control of the routebox component and is **JVM bound**. * strategy based indirection - direct payloads sent to the routebox endpoint along a camel route to specific inner routes based on a user defined internal routing strategy or a dispatch map. * exchange propagation - forward exchanges modified by the routebox endpoint to the next segment of the camel route.

The routebox component supports both consumer and producer endpoints.

Producer endpoints are of two flavors * Producers that send or dispatch incoming requests to a external routebox consumer endpoint * Producers that directly invoke routes in an internal embedded camel context thereby not sending requests to an external consumer.

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

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

The need for a Camel Routebox endpoint

The routebox component is designed to ease integration in complex environments needing * a large collection of routes and * involving a wide set of endpoint technologies needing integration in different ways

In such environments, it is often necessary to craft an integration solution by creating a sense of layering among camel routes effectively organizing them into * Coarse grained or higher level routes - aggregated collection of inner or lower level routes exposed as Routebox endpoints that represent an integration focus area. For example ||Focus Area||Coarse grained Route Examples|| |Department Focus|HR routes, Sales routes

etc| |Supply chain & B2B Focus|Shipping routes, Fulfillment routes, 3rd party services etc| |Technology Focus|Database routes, JMS routes, Scheduled batch routes etc| * Fine grained routes - routes that execute a singular and specific business and/or integration pattern.

Requests sent to Routebox endpoints on coarse grained routes can then delegate requests to inner fine grained routes to achieve a specific integration objective, collect the final inner result, and continue to progress to the next step along the coarse-grained route.

URI format

```
routebox:routeboxname[?options]
```

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

Options

Name	Default Value	Description
`dispatchStrategy`	`null`	A string representing a key in the Camel Registry matching an object value implementing the interface *org.apache.camel.component.routebox.strategy.RouteboxDispatchStrategy*
`dispatchMap`	`null`	A string representing a key in the Camel Registry matching an object value of the type HashMap<String, String>. The HashMap key should contain strings that can be matched against the value set for the exchange header **ROUTE_DISPATCH_KEY**. The HashMap value should contain inner route consumer URI's to which requests should be directed.
`innerContext`	`auto created`	A string representing a key in the Camel Registry matching an object value of the type *org.apache.camel.CamelContext*. If a CamelContext is not provided by the user a CamelContext is automatically created for deployment of inner routes.
`innerRegistry`	`null`	A string representing a key in the Camel Registry matching an object value that implements the interface *org.apache.camel.spi.Registry*. If Registry values are utilized by inner routes to create endpoints, an innerRegistry parameter must be provided
`routeBuilders`	`empty List`	A string representing a key in the Camel Registry matching an object value of the type *List<org.apache.camel.builder.RouteBuilder>*. If the user does not supply an innerContext pre-primed with inner routes, the routeBuilders option must be provided as a non-empty list of RouteBuilders containing inner routes

`innerProtocol`	`Direct`	The Protocol used internally by the Routebox component. Can be Direct or SEDA. **The Routebox component currently offers protocols that are JVM bound.**
`sendToConsumer`	`true`	Dictates whether a Producer endpoint sends a request to an external routebox consumer. If the setting is false, the Producer creates an embedded inner context and processes requests internally.
`forkContext`	`true`	The Protocol used internally by the Routebox component. Can be Direct or SEDA. **The Routebox component currently offers protocols that are JVM bound.**
`threads`	`20`	Number of threads to be used by the routebox to receive requests. **Setting applicable only for innerProtocol SEDA**.
`queueSize`	`unlimited`	Create a fixed size queue to receive requests. **Setting applicable only for innerProtocol SEDA**.

Sending/Receiving Messages to/from the routebox

Before sending requests it is necessary to properly configure the routebox by loading the required URI parameters into the Registry as shown below. In the case of Spring, if the necessary beans are declared correctly, the registry is automatically populated by Camel.

Step 1: Loading inner route details into the Registry

```
@Override
protected JndiRegistry createRegistry() throws Exception {
    JndiRegistry registry = new JndiRegistry(createJndiContext());

    // Wire the routeDefinitions & dispatchStrategy to the outer camelContext where the
routebox is declared
    List<RouteBuilder> routes = new ArrayList<RouteBuilder>();
    routes.add(new SimpleRouteBuilder());
    registry.bind("registry", createInnerRegistry());
    registry.bind("routes", routes);

    // Wire a dispatch map to registry
    HashMap<String, String> map = new HashMap<String, String>();
    map.put("addToCatalog", "seda:addToCatalog");
    map.put("findBook", "seda:findBook");
    registry.bind("map", map);

    // Alternatively wiring a dispatch strategy to the registry
    registry.bind("strategy", new SimpleRouteDispatchStrategy());
```

```
    return registry;
}

private JndiRegistry createInnerRegistry() throws Exception {
    JndiRegistry innerRegistry = new JndiRegistry(createJndiContext());
    BookCatalog catalogBean = new BookCatalog();
    innerRegistry.bind("library", catalogBean);

    return innerRegistry;
}
...
CamelContext context = new DefaultCamelContext(createRegistry());
```

Step 2: Optionaly using a Dispatch Strategy instead of a Dispatch Map

Using a dispatch Strategy involves implementing the interface *org.apache.camel.component.routebox.strategy.RouteboxDispatchStrategy* as shown in the example below.

```
public class SimpleRouteDispatchStrategy implements RouteboxDispatchStrategy {

    /* (non-Javadoc)
     * @see org.apache.camel.component.routebox.strategy.RouteboxDispatchStrategy#selectDes
tinationUri(java.util.List, org.apache.camel.Exchange)
     */
    public URI selectDestinationUri(List<URI> activeDestinations,
            Exchange exchange) {
        URI dispatchDestination = null;

        String operation = exchange.getIn().getHeader("ROUTE_DISPATCH_KEY", String.class);

        for (URI destination : activeDestinations) {
            if (destination.toASCIIString().equalsIgnoreCase("seda:" + operation)) {
                dispatchDestination = destination;
                break;
            }
        }

        return dispatchDestination;
    }
}
```

Step 2: Launching a routebox consumer

When creating a route consumer, note that the # entries in the routeboxUri are matched to the created inner registry, routebuilder list and dispatchStrategy/dispatchMap in the CamelContext Registry. Note that all routebuilders and associated routes are launched in the routebox created inner context

```
private String routeboxUri = "routebox:multipleRoutes?innerRegistry=#registry&routeBuild
ers=#routes&dispatchMap=#map";

public void testRouteboxRequests() throws Exception {
    CamelContext context = createCamelContext();
    template = new DefaultProducerTemplate(context);
    template.start();

    context.addRoutes(new RouteBuilder() {
        public void configure() {
            from(routeboxUri)
                .to("log:Routes operation performed?showAll=true");
        }
    });
    context.start();

    // Now use the ProducerTemplate to send the request to the routebox
   template.requestBodyAndHeader(routeboxUri, book, "ROUTE_DISPATCH_KEY", "addToCatalog");
}
```

Step 3: Using a routebox producer

When sending requests to the routebox, it is not necessary for producers do not need to know the inner route endpoint URI and they can simply invoke the Routebox URI endpoint with a dispatch strategy or dispatchMap as shown below

It is necessary to set a special exchange Header called **ROUTE_DISPATCH_KEY** (optional for Dispatch Strategy) with a key that matches a key in the dispatch map so that the request can be sent to the correct inner route

```
from("direct:sendToStrategyBasedRoutebox")
    .to("routebox:multipleRoutes?innerRegistry=#registry&routeBuilders=#routes&dispatch
Strategy=#strategy")
    .to("log:Routes operation performed?showAll=true");

from ("direct:sendToMapBasedRoutebox")
    .setHeader("ROUTE_DISPATCH_KEY", constant("addToCatalog"))
```

```
    .to("routebox:multipleRoutes?innerRegistry=#registry&routeBuilders=#routes&dis
patchMap=#map")
    .to("log:Routes operation performed?showAll=true");
```

Chapter 72. RSS

RSS Component

The **rss:** component is used for polling RSS feeds. Fuse Mediation Router will default poll the feed every 60th seconds.

Note: The component currently only supports polling (consuming) feeds.

URI format

```
rss:rssUri
```

Where `rssUri` is the URI to the RSS feed to poll.

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

Options

Property	Default	Description
`splitEntries`	`true`	If `true`, Fuse Mediation Router splits a feed into its individual entries and returns each entry, poll by poll. For example, if a feed contains seven entries, Fuse Mediation Router returns the first entry on the first poll, the second entry on the second poll, and so on. When no more entries are left in the feed, Fuse Mediation Router contacts the remote RSS URI to obtain a new feed. If `false`, Fuse Mediation Router obtains a fresh feed on every poll and returns all of the feed's entries.
`filter`	`true`	Use in combination with the `splitEntries` option in order to filter returned entries. By default, Fuse Mediation Router applies the `UpdateDateFilter` filter, which returns only new entries from the feed, ensuring that the consumer endpoint never receives an entry more than once. The filter orders the entries chronologically, with the newest returned last.
`throttleEntries`	`true`	**Camel 2.5:** Sets whether all entries identified in a single feed poll should be delivered immediately. If true, only one entry is processed per consumer.delay. Only applicable when splitEntries is set to true.
`lastUpdate`	`null`	Use in combination with the `filter` option to block entries earlier than a specific date/time (uses the `entry.updated` timestamp). The format is: `yyyy-MM-ddTHH:MM:ss`. Example: `2007-12-24T17:45:59`.
`feedHeader`	`true`	Specifies whether to add the ROME `SyndFeed` object as a header.

`sortEntries`	`false`	If `splitEntries` is `true`, this specifies whether to sort the entries by updated date.
`consumer.delay`	`60000`	Delay in milliseconds between each poll.
`consumer.initialDelay`	`1000`	Milliseconds before polling starts.
`consumer.userFixedDelay`	`false`	Set to `true` to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService[1] in JDK for details.

Exchange data types

Fuse Mediation Router initializes the In body on the Exchange with a ROME `SyndFeed`. Depending on the value of the `splitEntries` flag, Fuse Mediation Router returns either a `SyndFeed` with one `SyndEntry` or a `java.util.List` of `SyndEntrys`.

Option	Value	Behavior
`splitEntries`	`true`	A single entry from the current feed is set in the exchange.
`splitEntries`	`false`	The entire list of entries from the current feed is set in the exchange.

Message Headers

Header	Description
`org.apache.camel.component.rss.feed`	Fuse Mediation Router 1.x: The entire `SyncFeed` object.
`CamelRssFeed`	Fuse Mediation Router 2.0: The entire `SyncFeed` object.

RSS Dataformat

The RSS component ships with an RSS dataformat that can be used to convert between String (as XML) and ROME RSS model objects.

- marshal = from ROME `SyndFeed` to XML `String`
- unmarshal = from XML `String` to ROME `SyndFeed`

A route using this would look something like this:

```
from("rss:file:src/test/data/rss20.xml?splitEntries=false&consumer.delay=1000").mar
shal().rss().to("mock:marshal");
```

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html

The purpose of this feature is to make it possible to use Fuse Mediation Router's lovely built-in expressions for manipulating RSS messages. As shown below, an XPath expression can be used to filter the RSS message:

```
// only entries with Fuse Mediation Router in the title will get through the filter
from("rss:file:src/test/data/rss20.xml?splitEntries=true&consumer.delay=100")
    .marshal().rss().filter().xpath("//item/title[contains(.,'Camel')]").to("mock:result");
```

Filtering entries

You can filter out entries quite easily using XPath, as shown in the data format section above. You can also exploit Fuse Mediation Router's Bean Integration[2] to implement your own conditions. For instance, a filter equivalent to the XPath example above would be:

```
// only entries with Camel in the title will get through the filter
from("rss:file:src/test/data/rss20.xml?splitEntries=true&consumer.delay=100").
    filter().method("myFilterBean", "titleContainsCamel").to("mock:result");
```

The custom bean for this would be:

```
public static class FilterBean {
    public boolean titleContainsCamel(@Body SyndFeed feed) {
        SyndEntry firstEntry = (SyndEntry) feed.getEntries().get(0);
        return firstEntry.getTitle().contains("Camel");
    }
}
```

See also

- Atom on page 35

[2] Bean Integration

Chapter 73. Scalate

Scalate

Available as of Fuse Mediation Router 2.3

The **scalate:** component allows you to process a message using Scalate[1] template, which supports either SSP or Scaml format templates. This can be ideal when using Templating to generate responses for requests.

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

```
<dependency>
    <groupId>org.fusesource.scalate</groupId>
    <artifactId>scalate-camel</artifactId>
    <version>1.0</version>
</dependency>
```

URI format

```
scalate:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (eg: file://folder/myfile.ssp).

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

Message Headers

The scalate component sets a couple headers on the message (you can't set these yourself and from Fuse Mediation Router 2.1 scalate component will not set these headers which will cause some side effect on the dynamic template support):

Header	Description
`CamelScalateResource`	The resource as an `org.springframework.core.io.Resource` object.
`CamelScalateResourceUri`	The **templateName** as a `String` object.

Headers set during the Scalate evaluation are returned to the message and added as headers. Then its kinda possible to return values from Scalate to the Message.

For example, to set the header value of `fruit` in the Scalate template `.tm`:

[1] http://scalate.fusesource.org/

```
<% in.setHeader('fruit', 'Apple') %>
```

The `fruit` header is now accessible from the `message.out.headers`.

Scalate Context

Fuse Mediation Router will provide exchange information in the Scalate context (just a `Map`). The `Exchange` is transfered as:

key	value
`exchange`	The `Exchange` itself.
`headers`	The headers of the In message.
`camelContext`	The Camel Context intance.
`request`	The In message.
`in`	The In message.
`body`	The In message body.
`out`	The Out message (only for InOut message exchange pattern).
`response`	The Out message (only for InOut message exchange pattern).

Hot reloading

The Scalate template resource is, by default, hot reloadable for both file and classpath resources (expanded jar).

Dynamic templates

Fuse Mediation Router provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Fuse Mediation Router uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

Header	Type	Description
CamelScalateResourceUri	String	An URI for the template resource to use instead of the endpoint configured.
CamelScalateTemplate	String	The template to use instead of the endpoint configured.

Samples

For example you could use something like

```
from("activemq:My.Queue").
  to("scalate:com/acme/MyResponse.ssp");
```

To use a Scalate template to formulate a response to a message for InOut message exchanges (where there is a `JMSReplyTo` header).

If you want to use InOnly and consume the message and send it to another destination, you could use the following route:

```
from("activemq:My.Queue").
  to("scalate:com/acme/MyResponse.scaml").
  to("activemq:Another.Queue");
```

It's possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").
  setHeader("CamelScalateResourceUri").constant("path/to/my/template.scaml").
  to("scalate:dummy");
```

It's possible to specify a template directly as a header the component should use dynamically via a header, so for example:

```
from("direct:in").
  setHeader("CamelScalateTemplate").constant("<%@ attribute body: Object %>\nHi this is a
scalate template that can do templating ${body}").
  to("scalate:dummy");
```

The Email Sample

In this sample we want to use Scalate templating for an order confirmation email. The email template is laid out in Scalate as:

```
<%@ attribute in: org.apache.camel.scala.RichMessage %>
Dear ${in("lastName"}, ${in("firstName")}

Thanks for the order of ${in("item")}.

Regards Camel Riders Bookstore
${in.body}
```

Chapter 74. SEDA

SEDA Component

The **seda:** component provides asynchronous SEDA[1] behavior, so that messages are exchanged on a BlockingQueue[2] and consumers are invoked in a separate thread from the producer.

Note that queues are only visible within a *single* CamelContext. If you want to communicate across `CamelContext` instances (for example, communicating between Web applications), see the VM on page 519 component.

This component does not implement any kind of persistence or recovery, if the VM terminates while messages are yet to be processed. If you need persistence, reliability or distributed SEDA, try using either JMS on page 293 or ActiveMQ on page 25.

Synchronous

The Direct on page 111 component provides synchronous invocation of any consumers when a producer sends a message exchange.

URI format

```
seda:queueName[?options]
```

Where `queueName` can be any string that uniquely identifies the endpoint within the current CamelContext.

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

Note

When matching consumer entpoints to producer endpoints, only the `queueName` is considered and any option settings are ignored. That is, the identity of a consumer endpoint depends only on the `queueName`. If you want to attach multiple consumers to the same queue, use the approach described in "Using multipleConsumers" on page 432.

[1] http://www.eecs.harvard.edu/~mdw/proj/seda/
[2] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/BlockingQueue.html

Options

Name	Default	Description
`size`	*Unbounded*	The maximum size (= capacity of the number of messages it can max hold) of the SEDA queue. The default value in Camel 2.2 or older is `1000`. From Camel 2.3 onwards the size is unbounded by default.
`concurrentConsumers`	`1`	**Fuse Mediation Router 1.6.1/2.0**: Number of concurrent threads processing exchanges.
`waitForTaskToComplete`	`IfReplyExpected`	**Fuse Mediation Router 2.0**: Option to specify whether the caller should wait for the async task to complete or not before continuing. The following three options are supported: `Always`, `Never` or `IfReplyExpected`. The first two values are self-explanatory. The last value, `IfReplyExpected`, will only wait if the message is Request Reply based. The default option is `IfReplyExpected`. See more information about Async messaging.
`timeout`	`30000`	**Fuse Mediation Router 2.0:** Timeout in millis a seda producer will at most waiting for an async task to complete. See `waitForTaskToComplete` and Async for more details. In **Camel 2.2** you can now disable timeout by using 0 or a negative value.
`multipleConsumers`	`false`	**Camel 2.2:** Specifies whether multiple consumers are allowed or not. If enabled, you can use SEDA on page 429 for a publish/subscribe style of messaging. Send a message to a SEDA queue and have multiple consumers receive a copy of the message.
`limitConcurrentConsumers`	`true`	**Camel 2.3:** Whether to limit the concurrentConsumers to maximum 500. If its configured with a higher number an exception will be thrown. You can disable this check by turning this option off.

Changes in Fuse Mediation Router 2.0

In Fuse Mediation Router 2.0 the Seda component supports using Request Reply, where the caller will wait for the Async route to complete. For instance:

```
from("mina:tcp://0.0.0.0:9876?textline=true&sync=true").to("seda:input");

from("seda:input").to("bean:processInput").to("bean:createResponse");
```

In the route above, we have a TCP listener on port 9876 that accepts incoming requests. The request is routed to the `seda:input` queue. As it is a Request Reply message, we wait for the response. When the consumer on the `seda:input` queue is complete, it copies the response to the original message response.

Fuse Mediation Router 1.x does **not** have this feature implemented, the Seda queues in Fuse Mediation Router 1.x will never wait.

Camel 2.0 - 2.2: Works only with 2 endpoints

Using Request Reply over SEDA on page 429 or VM on page 519 only works with 2 endpoints. You **cannot** chain endpoints by sending to A -> B -> C etc. Only between A -> B. The reason is the implementation logic is fairly simple. To support 3+ endpoints makes the logic much more complex to handle ordering and notification between the waiting threads properly.

This has been improved in **Camel 2.3** onwards, which allows you to chain as many endpoints as you like.

Concurrent consumers

By default, the SEDA endpoint uses a single consumer thread, but you can configure it to use concurrent consumer threads. So instead of thread pools you can use:

```
from("seda:stageName?concurrentConsumers=5").process(...)
```

Difference between thread pools and concurrent consumers

The *thread pool* is a pool that can increase/shrink dynamically at runtime depending on load, whereas the concurrent consumers are always fixed.

Thread pools

Be aware that adding a thread pool to a SEDA endpoint by doing something like:

```
from("seda:stageName").thread(5).process(...)
```

Can wind up with two `BlockQueues`: one from the SEDA endpoint, and one from the workqueue of the thread pool, which may not be what you want. Instead, you might want to consider configuring a Direct on page 111 endpoint with a thread pool, which can process messages both synchronously and asynchronously. For example:

```
from("direct:stageName").thread(5).process(...)
```

You can also directly configure number of threads that process messages on a SEDA endpoint using the `concurrentConsumers` option.

Sample

In the route below we use the SEDA queue to send the request to this async queue to be able to send a fire-and-forget message for further processing in another thread, and return a constant reply in this thread to the original caller.

```
public void configure() throws Exception {
    from("direct:start")
        // send it to the seda queue that is async
        .to("seda:next")
        // return a constant response
        .transform(constant("OK"));

    from("seda:next").to("mock:result");
}
```

Here we send a Hello World message and expect the reply to be OK.

```
Object out = template.requestBody("direct:start", "Hello World");
assertEquals("OK", out);
```

The "Hello World" message will be consumed from the SEDA queue from another thread for further processing. Since this is from a unit test, it will be sent to a `mock` endpoint where we can do assertions in the unit test.

Using multipleConsumers

Available as of Camel 2.2

In this example we have defined two consumers and registered them as spring beans.

```
<!-- define the consumers as spring beans -->
<bean id="consumer1" class="org.apache.camel.spring.example.FooEventConsumer"/>

<bean id="consumer2" class="org.apache.camel.spring.example.AnotherFooEventConsumer"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- define a shared endpoint which the consumers can refer to instead of using url --
>
    <endpoint id="foo" uri="seda:foo?multipleConsumers=true"/>
</camelContext>
```

Since we have specified **multipleConsumers=true** on the seda foo endpoint we can have those two consumers receive their own copy of the message as a kind of pub-sub style messaging.

As the beans are part of an unit test they simply send the message to a mock endpoint, but notice how we can use @Consume to consume from the seda queue.

```
public class FooEventConsumer {

    @EndpointInject(uri = "mock:result")
    private ProducerTemplate destination;

    @Consume(ref = "foo")
    public void doSomething(String body) {
        destination.sendBody("foo" + body);
    }

}
```

See Also

- VM on page 519
- Direct on page 111

Chapter 75. SERVLET

Servlet Component

The **servlet:** component provides HTTP based endpoints[1] for consuming HTTP requests that arrive at a HTTP endpoint and this endpoint is bound to a published Servlet.

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

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

URI format

```
servlet://relative_path[?options]
```

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

Options

Name	Default Value	Description
`httpBindingRef`	`null`	Reference to an `org.apache.camel.component.http.HttpBinding` in the Registry. A `HttpBinding` implementation can be used to customize how to write a response.
`matchOnUriPrefix`	`false`	Whether or not the `CamelServlet` should try to find a target consumer by matching the URI prefix, if no exact match is found.
`servletName`	`null`	Specifies the servlet name that the servlet endpoint will bind to. If there is no servlet name specified, the servlet endpoint will be bind to first published Servlet

Message Headers

Fuse Mediation Router will apply the same Message Headers as the HTTP on page 229 component.

[1] Endpoint

Fuse Mediation Router will also populate **all**`request.parameter` and `request.headers`. For example, if a client request has the URL, `http://myserver/myserver?orderid=123`, the exchange will contain a header named `orderid` with the value 123.

Usage

You can only consume from endpoints generated by the Servlet component. Therefore, it should only be used as input into your Fuse Mediation Router routes. To issue HTTP requests against other HTTP endpoints, use the HTTP Component on page 229

Sample

In this sample, we define a route that exposes a HTTP service at `http://localhost:8080/camel/services/hello`. First, you need to publish the CamelHttpTransportServlet[2] through the normal Web Container, or OSGi Service. Use the `Web.xml` file to publish the CamelHttpTransportServlet[3] as follows:

```
<web-app>

  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <display-name>Camel Http Transport Servlet</display-name>
    <servlet-class>
        org.apache.camel.component.servlet.CamelHttpTransportServlet
    </servlet-class>

  </servlet>

  <servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Use an `Activator` to publish the CamelHttpTransportServlet[4] on the OSGi platform

```
import java.util.Dictionary;
import java.util.Hashtable;

import javax.servlet.Servlet;
```

[2] http://svn.apache.org/repos/asf/camel/trunk/components/camel-servlet/src/main/java/org/apache/camel/component/servlet/CamelHttpTransportServlet.java

[3] http://svn.apache.org/repos/asf/camel/trunk/components/camel-servlet/src/main/java/org/apache/camel/component/servlet/CamelHttpTransportServlet.java

[4] http://svn.apache.org/repos/asf/camel/trunk/components/camel-servlet/src/main/java/org/apache/camel/component/servlet/CamelHttpTransportServlet.java

```
import org.apache.camel.component.servlet.CamelHttpTransportServlet;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import org.osgi.service.http.HttpContext;
import org.osgi.service.http.HttpService;
import org.springframework.osgi.context.BundleContextAware;

public final class ServletActivator implements BundleActivator, BundleContextAware {
    private static final transient Log LOG = LogFactory.getLog(ServletActivator.class);
    private static boolean registerService;

    /**
     * HttpService reference.
     */
    private ServiceReference httpServiceRef;

    /**
     * Called when the OSGi framework starts our bundle
     */
    public void start(BundleContext bc) throws Exception {
        registerServlet(bc);
    }

    /**
     * Called when the OSGi framework stops our bundle
     */
    public void stop(BundleContext bc) throws Exception {
        if (httpServiceRef != null) {
            bc.ungetService(httpServiceRef);
            httpServiceRef = null;
        }
    }

    protected void registerServlet(BundleContext bundleContext) throws Exception {
        httpServiceRef = bundleContext.getServiceReference(HttpService.class.getName());

        if (httpServiceRef != null && !registerService) {
            LOG.info("Regist the servlet service");
            final HttpService httpService = (HttpService)bundleContext.getService(httpSer
viceRef);
            if (httpService != null) {
                // create a default context to share between registrations
                final HttpContext httpContext = httpService.createDefaultHttpContext();
                // register the hello world servlet
              final Dictionary<String, String> initParams = new Hashtable<String, String>();
```

```
                initParams.put("matchOnUriPrefix", "false");
                initParams.put("servlet-name", "camelServlet");
                httpService.registerServlet("/camel/services", // alias
                    (Servlet)new CamelHttpTransportServlet(), // register servlet
                    initParams, // init params
                    httpContext // http context
                );
                registerService = true;
            }
        }
    }

    public void setBundleContext(BundleContext bc) {
        try {
            registerServlet(bc);
        } catch (Exception e) {
            LOG.error("Can't register the servlet, the reason is " + e);
        }
    }

}
```

Then you can define your route as follows:

```
from("servlet:///hello?matchOnUriPrefix=true").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
       String contentType = exchange.getIn().getHeader(Exchange.CONTENT_TYPE, String.class);

        String path = exchange.getIn().getHeader(Exchange.HTTP_PATH, String.class);
        assertEquals("Get a wrong content type", CONTENT_TYPE, contentType);
        // assert camel http header
       String charsetEncoding = exchange.getIn().getHeader(Exchange.HTTP_CHARACTER_ENCODING,
 String.class);
       assertEquals("Get a wrong charset name from the message header", "UTF-8", charsetEn
coding);
        // assert exchange charset
        assertEquals("Get a wrong charset naem from the exchange property", "UTF-8", ex
change.getProperty(Exchange.CHARSET_NAME));
        exchange.getOut().setHeader(Exchange.CONTENT_TYPE, contentType + "; charset=UTF-
8");
        exchange.getOut().setHeader("PATH", path);
        exchange.getOut().setBody("<b>Hello World</b>");
    }
    });
```

From **Camel 2.6.0**, you can also publish the CamelHttpTransportServlet[5] as an OSGi service with help of SpringDM like this.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:osgi="http://www.springframework.org/schema/osgi"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/osgi  http://www.springframe
work.org/schema/osgi/spring-osgi.xsd">

  <bean id="osgiServlet" class="org.apache.camel.component.servlet.CamelHttpTransportSer
vlet"/>

  <osgi:service ref="osgiServlet">
    <osgi:interfaces>
      <value>javax.servlet.Servlet</value>
      <value>org.apache.camel.component.servlet.CamelServletService</value>
    </osgi:interfaces>
    <osgi:service-properties>
      <entry key="alias" value="/camel/services" />
      <entry key="servlet-name" value="CamelServlet"/>
    </osgi:service-properties>
  </osgi:service>

</beans>
```

Then use this service in your camel route like this:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:camel="http://camel.apache.org/schema/spring"
  xmlns:osgi="http://www.springframework.org/schema/osgi"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/osgi  http://www.springframe
work.org/schema/osgi/spring-osgi.xsd
    http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <osgi:reference id="camelHttpTransportServlet" interface="org.apache.camel.component.ser
vlet.CamelServletService"/>

  <bean id="servlet" class="org.apache.camel.component.servlet.ServletComponent">
```

5 http://svn.apache.org/repos/asf/camel/trunk/components/camel-servlet/src/main/java/org/apache/camel/component/servlet/CamelHttpTransportServlet.java

```
    <property name = "camelServletService" ref="camelHttpTransportServlet" />
  </bean>

  <bean id="servletProcessor" class="org.apache.camel.itest.osgi.servlet.ServletProcessor"
 />

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <camel:route>
      <camel:from uri="servlet:///hello"/>
      <camel:process ref="servletProcessor"/>
    </camel:route>
  </camelContext>

</beans>
```

Specify the relative path for camel-servlet endpoint

Since we are binding the HTTP transport with a published servlet, and we don't know the servlet's application context path, the `camel-servlet` endpoint uses the relative path to specify the endpoint's URL. A client can access the `camel-servlet` endpoint through the servlet publish address: `("http://localhost:8080/camel/services") + RELATIVE_PATH("/hello")`.

Chapter 76. Shiro Security

Shiro Security Component

Available as of Camel 2.5

The **shiro-security** component in Camel is a security focused component, based on the Apache Shiro security project.

Apache Shiro is a powerful and flexible open-source security framework that cleanly handles authentication, authorization, enterprise session management and cryptography. The objective of the Apache Shiro project is to provide the most robust and comprehensive application security framework available while also being very easy to understand and extremely simple to use.

This camel shiro-security component allows authentication and authorization support to be applied to different segments of a camel route.

Shiro security is applied on a route using a Camel Policy. A Policy in Camel utilizes a strategy pattern for applying interceptors on Camel Processors. It offering the ability to apply cross-cutting concerns (for example. security, transactions etc) on sections/segments of a camel route.

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

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

Shiro Security Basics

To employ Shiro security on a camel route, a ShiroSecurityPolicy object must be instantiated with security configuration details (including users, passwords, roles etc). This object must then be applied to a camel route. This ShiroSecurityPolicy Object may also be registered in the Camel registry (JNDI or ApplicationContextRegistry) and then utilized on other routes in the Camel Context.

Configuration details are provided to the ShiroSecurityPolicy using an Ini file (properties file) or an Ini object. The Ini file is a standard Shiro configuration file containing user/role details as shown below

```
[users]
```

```
# user 'ringo' with password 'starr' and the 'sec-level1' role
ringo = starr, sec-level1
george = harrison, sec-level2
john = lennon, sec-level3
paul = mccartney, sec-level3

[roles]
# 'sec-level3' role has all permissions, indicated by the
# wildcard '*'
sec-level3 = *

# The 'sec-level2' role can do anything with access of permission
# readonly (*) to help
sec-level2 = zone1:*

# The 'sec-level1' role can do anything with access of permission
# readonly
sec-level1 = zone1:readonly:*
```

Instantiating a ShiroSecurityPolicy Object

A ShiroSecurityPolicy object is instantiated as follows

```
        private final String iniResourcePath = "classpath:shiro.ini";
        private final byte[] passPhrase = {
            (byte) 0x08, (byte) 0x09, (byte) 0x0A, (byte) 0x0B,
            (byte) 0x0C, (byte) 0x0D, (byte) 0x0E, (byte) 0x0F,
            (byte) 0x10, (byte) 0x11, (byte) 0x12, (byte) 0x13,
            (byte) 0x14, (byte) 0x15, (byte) 0x16, (byte) 0x17};
        List<permission> permissionsList = new ArrayList<permission>();
        Permission permission = new WildcardPermission("zone1:readwrite:*");
        permissionsList.add(permission);

        final ShiroSecurityPolicy securityPolicy =
            new ShiroSecurityPolicy(iniResourcePath, passPhrase, true, permissionsList);
```

ShiroSecurityPolicy Options

Name	Default Value	Type	Description
`iniResourcePath or ini`	`none`	Resource String or Ini Object	A mandatory Resource String for the iniResourcePath or an instance

			of an Ini object must be passed to the security policy. Resources can be acquired from the file system, classpath, or URLs when prefixed with "file:, classpath:, or url:" respectively. For e.g "classpath:shiro.ini"
passPhrase	An AES 128 based key	byte[]	A passPhrase to decrypt ShiroSecurityToken(s) sent along with Message Exchanges
alwaysReauthenticate	true	boolean	Setting to ensure re-authentication on every individual request. If set to false, the user is authenticated and locked such than only requests from the same user going forward are authenticated.
permissionsList	none	List<Permission>	A List of permissions required in order for an authenticated user to be authorized to perform further action i.e continue further on the route. If no Permissions list is provided to the ShiroSecurityPolicy object, then authorization is deemed as not required
cipherService	AES	org.apache.shiro.crypto.CipherService	Shiro ships with AES & Blowfish based CipherServices. You may use one these or pass in your own Cipher implementation

Applying Shiro Authentication on a Camel Route

The ShiroSecurityPolicy, tests and permits incoming message exchanges containing a encrypted SecurityToken in the Message Header to proceed further following proper authentication. The SecurityToken object contains a Username/Password details that are used to determine where the user is a valid user.

```
    protected RouteBuilder createRouteBuilder() throws Exception {
        final ShiroSecurityPolicy securityPolicy =
            new ShiroSecurityPolicy("classpath:shiro.ini", passPhrase);

        return new RouteBuilder() {
```

```
            public void configure() {
                onException(UnknownAccountException.class).
                    to("mock:authenticationException");
                onException(IncorrectCredentialsException.class).
                    to("mock:authenticationException");
                onException(LockedAccountException.class).
                    to("mock:authenticationException");
                onException(AuthenticationException.class).
                    to("mock:authenticationException");

                from("direct:secureEndpoint").
                    to("log:incoming payload").
                    policy(securityPolicy).
                    to("mock:success");
            }
        };
    }
```

Applying Shiro Authorization on a Camel Route

Authorization can be applied on a camel route by associating a Permissions List with the ShiroSecurityPolicy. The Permissions List specifies the permissions necessary for the user to proceed with the execution of the route segment. If the user does not have the proper permission set, the request is not authorized to continue any further.

```
    protected RouteBuilder createRouteBuilder() throws Exception {
        final ShiroSecurityPolicy securityPolicy =
           new ShiroSecurityPolicy("./src/test/resources/securityconfig.ini", passPhrase);


        return new RouteBuilder() {
            public void configure() {
                onException(UnknownAccountException.class).
                    to("mock:authenticationException");
                onException(IncorrectCredentialsException.class).
                    to("mock:authenticationException");
                onException(LockedAccountException.class).
                    to("mock:authenticationException");
                onException(AuthenticationException.class).
                    to("mock:authenticationException");

                from("direct:secureEndpoint").
                    to("log:incoming payload").
                    policy(securityPolicy).
                    to("mock:success");
            }
```

```
        };
    }
```

Creating a ShiroSecurityToken and injecting it into a Message Exchange

A ShiroSecurityToken object may be created and injected into a Message Exchange using a Shiro Processor called ShiroSecurityTokenInjector. An example of injecting a ShiroSecurityToken using a ShiroSecurityTokenInjector in the client is shown below

```
ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "starr");
ShiroSecurityTokenInjector shiroSecurityTokenInjector =
    new ShiroSecurityTokenInjector(shiroSecurityToken, passPhrase);

from("direct:client").
    process(shiroSecurityTokenInjector).
    to("direct:secureEndpoint");
```

Sending Messages to routes secured by a ShiroSecurityPolicy

Messages and Message Exchanges sent along the camel route where the security policy is applied need to be accompanied by a SecurityToken in the Exchange Header. The SecurityToken is an encrypted object that holds a Username and Password. The SecurityToken is encrypted using AES 128 bit security by default and can be changed to any cipher of your choice.

Given below is an example of how a request may be sent using a ProducerTemplate in Camel along with a SecurityToken

```
@Test
public void testSuccessfulShiroAuthenticationWithNoAuthorization() throws Exception {

    //Incorrect password
    ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "stirr");

    // TestShiroSecurityTokenInjector extends ShiroSecurityTokenInjector
    TestShiroSecurityTokenInjector shiroSecurityTokenInjector =
        new TestShiroSecurityTokenInjector(shiroSecurityToken, passPhrase);

    successEndpoint.expectedMessageCount(1);
    failureEndpoint.expectedMessageCount(0);

    template.send("direct:secureEndpoint", shiroSecurityTokenInjector);
```

```
        successEndpoint.assertIsSatisfied();
        failureEndpoint.assertIsSatisfied();
    }
```

Chapter 77. Sip

SIP Component

Available as of Camel 2.5

The **sip** component in Camel is a communication component, based on the Jain SIP implementation (available under the JCP license).

Session Initiation Protocol (SIP) is an IETF-defined signaling protocol, widely used for controlling multimedia communication sessions such as voice and video calls over Internet Protocol (IP).The SIP protocol is an Application Layer protocol designed to be independent of the underlying transport layer; it can run on Transmission Control Protocol (TCP), User Datagram Protocol (UDP) or Stream Control Transmission Protocol (SCTP).

The Jain SIP implementation supports TCP and UDP only.

The Camel SIP component **only** supports the SIP Publish and Subscribe capability as described in the RFC3903 - Session Initiation Protocol (SIP) Extension for Event[1]

This camel component supports both producer and consumer endpoints.

Camel SIP Producers (Event Publishers) and SIP Consumers (Event Subscribers) communicate event & state information to each other using an intermediary entity called a SIP Presence Agent (a stateful brokering entity).

For SIP based communication, a SIP Stack with a listener **must** be instantiated on both the SIP Producer and Consumer (using separate ports if using localhost). This is necessary in order to support the handshakes & acknowledgements exchanged between the SIP Stacks during communication.

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

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

[1] http://www.ietf.org/rfc/rfc3903.txt

URI format

The URI scheme for a sip endpoint is as follows:

```
sip://johndoe@localhost:99999[?options]
sips://johndoe@localhost:99999/[?options]
```

This component supports producer and consumer endpoints for both TCP and UDP.

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

Options

The SIP Component offers an extensive set of configuration options & capability to create custom stateful headers needed to propagate state via the SIP protocol.

Name	Default Value	Description
`stackName`	`NAME_NOT_SET`	Name of the SIP Stack instance associated with an SIP Endpoint.
`transport`	`tcp`	Setting for choice of transport potocol. Valid choices are "tcp" or "udp".
`fromUser`		Username of the message originator. **Mandatory setting unless a registry based custom FromHeader is specified.**
`fromHost`		Hostname of the message originator. **Mandatory setting unless a registry based FromHeader is specified**
`fromPort`		Port of the message originator. **Mandatory setting unless a registry based FromHeader is specified**
`toUser`		Username of the message receiver. **Mandatory setting unless a registry based custom ToHeader is specified.**
`toHost`		Hostname of the message receiver. **Mandatory setting unless a registry based ToHeader is specified**
`toPort`		Portname of the message receiver. **Mandatory setting unless a registry based ToHeader is specified**
`maxforwards`	0	the number of intermediaries that may forward the message to the message receiver. **Optional setting. May alternatively be set using as registry based MaxForwardsHeader**
`eventId`		Setting for a String based event Id. **Mandatory setting unless a registry based FromHeader is specified**

`eventHeaderName`		Setting for a String based event Id. **Mandatory setting unless a registry based FromHeader is specified**
`maxMessageSize`	`1048576`	Setting for maximum allowed Message size in bytes.
`cacheConnections`	`false`	Should connections be cached by the SipStack to reduce cost of connection creation. This is useful if the connection is used for long running conversations.
`consumer`	`false`	This setting is used to determine whether the kind of header (FromHeader,ToHeader etc) that needs to be created for this endpoint
`automaticDialogSupport`	`off`	Setting to specify whether every communication should be associated with a dialog.
`contentType`	`text`	Setting for contentType can be set to any valid MimeType.
`contentSubType`	`xml`	Setting for contentSubType can be set to any valid MimeSubType.
`receiveTimeoutMillis`	`10000`	Setting for specifying amount of time to wait for a Response and/or Acknowledgement can be received from another SIP stack
`useRouterForAllUris`	`false`	This setting is used when requests are sent to the Presence Agent via a proxy.
`msgExpiration`	`3600`	The amount of time a message received at an endpoint is considered valid
`presenceAgent`	`false`	This setting is used to distingish between a Presence Agent & a consumer. This is due to the fact that the SIP Camel component ships with a basic Presence Agent (for testing purposes only). Consumers have to set this flag to true.

Registry based Options

SIP requires a number of headers to be sent/received as part of a request. These SIP header can be enlisted in the Registry, such as in the Spring XML file.

The values that could be passed in, are the following:

Name	Description
`fromHeader`	a custom Header object containing message originator settings. Must implement the type javax.sip.header.FromHeader
`toHeader`	a custom Header object containing message receiver settings. Must implement the type javax.sip.header.ToHeader

`viaHeaders`	List of custom Header objects of the type javax.sip.header.ViaHeader. Each ViaHeader containing a proxy address for request forwarding. (Note this header is automatically updated by each proxy when the request arrives at its listener)
`contentTypeHeader`	a custom Header object containing message content details. Must implement the type javax.sip.header.ContentTypeHeader
`callIdHeader`	a custom Header object containing call details. Must implement the type javax.sip.header.CallIdHeader
`maxForwardsHeader`	a custom Header object containing details on maximum proxy forwards. This header places a limit on the viaHeaders possible. Must implement the type javax.sip.header.MaxForwardsHeader
`eventHeader`	a custom Header object containing event details. Must implement the type javax.sip.header.EventHeader
`contactHeader`	an optional custom Header object containing verbose contact details (email, phone number etc). Must implement the type javax.sip.header.ContactHeader
`expiresHeader`	a custom Header object containing message expiration details. Must implement the type javax.sip.header.ExpiresHeader
`extensionHeader`	a custom Header object containing user/application specific details. Must implement the type javax.sip.header.ExtensionHeader

Sending Messages to/from a SIP endpoint

Creating a Camel SIP Publisher

In the example below, a SIP Publisher is created to send SIP Event publications to a user "agent@localhost:5152". This is the address of the SIP Presence Agent which acts as a broker between the SIP Publisher and Subscriber

- using a SIP Stack named client
- using a registry based eventHeader called evtHdrName
- using a registry based eventId called evtId
- from a SIP Stack with Listener set up as user2@localhost:3534
- The Event being published is EVENT_A
- A Mandatory Header called REQUEST_METHOD is set to Request.Publish thereby setting up the endpoint as a Event publisher"

```
producerTemplate.sendBodyAndHeader(
    "sip://agent@localhost:5152?stackName=client&eventHeaderName=evtHdrName&eventId=evt
id&fromUser=user2&fromHost=localhost&fromPort=3534",
    "EVENT_A",
    "REQUEST_METHOD",
    Request.PUBLISH);
```

Creating a Camel SIP Subscriber

In the example below, a SIP Subscriber is created to receive SIP Event publications sent to a user "johndoe@localhost:5154"

- using a SIP Stack named Subscriber
- registering with a Presence Agent user called agent@localhost:5152
- using a registry based eventHeader called evtHdrName. The evtHdrName contains the Event which is se to "Event_A"
- using a registry based eventId called evtId

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            // Create PresenceAgent
          from("sip://agent@localhost:5152?stackName=PresenceAgent&presenceAgent=true&event
HeaderName=evtHdrName&eventId=evtid")
                .to("mock:neverland");

            // Create Sip Consumer(Event Subscriber)
           from("sip://johndoe@localhost:5154?stackName=Subscriber&toUser=agent&toHost=loc
alhost&toPort=5152&eventHeaderName=evtHdrName&eventId=evtid")
                .to("log:ReceivedEvent?level=DEBUG")
                .to("mock:notification");

        }
    };
}
```

The Camel SIP component also ships with a Presence Agent that is meant to be used for Testing and Demo purposes only. An example of instantiating a Presence Agent is given above.

Note that the Presence Agent is set up as a user agent@localhost:5152 and is capable of communicating with both Publisher as well as Subscriber. It has a separate SIP stackName distinct from Publisher as well as Subscriber. While it is set up as a Camel Consumer, it does not actually send any messages along the route to the endpoint "mock:neverland".

Chapter 78. Smooks

Smooks

The **smooks** component supports the Smooks Library[1] for EDI parsing. The **camel-smooks** library is provided by the Camel Extra[2] project which hosts all *GPL related components for Fuse Mediation Router.

It is **only** the EDI parsing feature that is implemented in this component. The other features from Smooks are covered in existing camel components. Parsing from any given data source to EDI is implemented using Fuse Mediation Router Data Format.

@Deprecated

The `camel-smooks` component is @deprecated as the Smooks Library will integrate Camel out of the box.

[1] http://milyn.codehaus.org/Smooks
[2] http://code.google.com/p/camel-extra/

Chapter 79. SMPP

SMPP Component

This component provides access to an SMSC (Short Message Service Center) over the SMPP[1] protocol to send and receive SMS. The JSMPP[2] is used.

Using Fuse Mediation Router 2.2 onwards

This component is only available for Fuse Mediation Router 2.2 or newer.

URI format

```
smpp://[username@]hostname[:port][?options]
smpps://[username@]hostname[:port][?options]
```

If no **username** is provided, then Fuse Mediation Router will provide the default value `smppclient`. If no **port** number is provided, then Fuse Mediation Router will provide the default value `2775`.**Camel 2.3:** If the protocol name is `smpps`, camel-smpp with try to use SSLSocket to init a connection to the server.

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

URI Options

Name	Default Value	Description
`password`	`password`	Specifies the password to use to log in to the SMSC.
`systemType`	`cp`	This parameter is used to categorize the type of ESME (External Short Message Entity) that is binding to the SMSC (max. 13 characters).
`dataCoding`	`0`	**Camel 2.5 onwards** Defines encoding of data according the SMPP 3.4 specification, section 5.2.19. Example data encodings are: `0`: SMSC Default Alphabet `4`: 8 bit Alphabet `8`: UCS2 Alphabet
`encoding`	`ISO-8859-1`	Defines the encoding scheme of the short message user data.

[1] http://smsforum.net/SMPP_v3_4_Issue1_2.zip
[2] http://code.google.com/p/jsmpp/

enquireLinkTimer	5000	Defines the interval in milliseconds between the confidence checks. The confidence check is used to test the communication path between an ESME and an SMSC.
transactionTimer	10000	Defines the maximum period of inactivity allowed after a transaction, after which an SMPP entity may assume that the session is no longer active. This timer may be active on either communicating SMPP entity (i.e. SMSC or ESME).
initialReconnectDelay	5000	Defines the initial delay in milliseconds after the consumer/producer tries to reconnect to the SMSC, after the connection was lost.
reconnectDelay	5000	Defines the interval in milliseconds between the reconnect attempts, if the connection to the SMSC was lost and the previous was not succeed.
registeredDelivery	1	Is used to request an SMSC delivery receipt and/or SME originated acknowledgements. The following values are defined: 0: No SMSC delivery receipt requested. 1: SMSC delivery receipt requested where final delivery outcome is success or failure. 2: SMSC delivery receipt requested where the final delivery outcome is delivery failure.
serviceType	CMT	The service type parameter can be used to indicate the SMS Application service associated with the message. The following generic service_types are defined: • CMT: Cellular Messaging • CPT: Cellular Paging • VMN: Voice Mail Notification • VMA: Voice Mail Alerting • WAP: Wireless Application Protocol • USSD: Unstructured Supplementary Services Data
sourceAddr	1616	Defines the address of SME (Short Message Entity) which originated this message.
destAddr	1717	Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
sourceAddrTon	0	Defines the type of number (TON) to be used in the SME originator address parameters. The following TON values are defined: • 0: Unknown

		• 1: International • 2: National • 3: Network Specific • 4: Subscriber Number • 5: Alphanumeric • 6: Abbreviated
destAddrTon	0	Defines the type of number (TON) to be used in the SME destination address parameters. The following TON values are defined: • 0: Unknown • 1: International • 2: National • 3: Network Specific • 4: Subscriber Number • 5: Alphanumeric • 6: Abbreviated
sourceAddrNpi	0	Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. The following NPI values are defined: • 0: Unknown • 1: ISDN (E163/E164) • 2: Data (X.121) • 3: Telex (F.69) • 6: Land Mobile (E.212) • 8: National • 9: Private

		• `10`: ERMES • `13`: Internet (IP) • `18`: WAP Client Id (to be defined by WAP Forum)
`destAddrNpi`	`0`	Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. The following NPI values are defined: • `0`: Unknown • `1`: ISDN (E163/E164) • `2`: Data (X.121) • `3`: Telex (F.69) • `6`: Land Mobile (E.212) • `8`: National • `9`: Private • `10`: ERMES • `13`: Internet (IP) • `18`: WAP Client Id (to be defined by WAP Forum)
`priorityFlag`	`1`	Allows the originating SME to assign a priority level to the short message. Four Priority Levels are supported: • `0`: Level 0 (lowest) priority • `1`: Level 1 priority • `2`: Level 2 priority • `3`: Level 3 (highest) priority
`replaceIfPresentFlag`	`0`	Used to request the SMSC to replace a previously submitted message, that is still pending delivery. The SMSC will replace an existing message provided that the source address, destination

		address and service type match the same fields in the new message. The following replace if present flag values are defined: • `0`: Don't replace • `1`: Replace
`dataCoding`	`0`	**Camel 2.5 onwarts** Defines encoding of data according the SMPP 3.4 specification, section 5.2.19. Example data encodings are: `0`: SMSC Default Alphabet `4`: 8 bit Alphabet `8`: UCS2 Alphabet
`typeOfNumber`	`0`	Defines the type of number (TON) to be used in the SME. The following TON values are defined: • `0`: Unknown • `1`: International • `2`: National • `3`: Network Specific • `4`: Subscriber Number • `5`: Alphanumeric • `6`: Abbreviated
`numberingPlanIndicator`	`0`	Defines the numeric plan indicator (NPI) to be used in the SME. The following NPI values are defined: • `0`: Unknown • `1`: ISDN (E163/E164) • `2`: Data (X.121) • `3`: Telex (F.69) • `6`: Land Mobile (E.212) • `8`: National • `9`: Private • `10`: ERMES

	• 13: Internet (IP) • 18: WAP Client Id (to be defined by WAP Forum)

You can have as many of these options as you like.

```
smpp://smppclient@localhost:2775?password=password&enquireLinkTimer=3000&transaction
Timer=5000&systemType=consumer
```

Message Headers

The following message headers can be used to affect the behavior of the SMPP producer

Header	Description
CamelSmppDestAddr	Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
CamelSmppDestAddrTon	Defines the type of number (TON) to be used in the SME destination address parameters. The following TON values are defined: • 0: Unknown • 1: International • 2: National • 3: Network Specific • 4: Subscriber Number • 5: Alphanumeric • 6: Abbreviated
CamelSmppDestAddrNpi	Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. The following NPI values are defined: • 0: Unknown • 1: ISDN (E163/E164) • 2: Data (X.121) • 3: Telex (F.69)

	• 6: Land Mobile (E.212) • 8: National • 9: Private • 10: ERMES • 13: Internet (IP) • 18: WAP Client Id (to be defined by WAP Forum)
CamelSmppSourceAddr	Defines the address of SME (Short Message Entity) which originated this message.
CamelSmppSourceAddrTon	Defines the type of number (TON) to be used in the SME originator address parameters. The following TON values are defined: • 0: Unknown • 1: International • 2: National • 3: Network Specific • 4: Subscriber Number • 5: Alphanumeric • 6: Abbreviated
CamelSmppSourceAddrNpi	Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. The following NPI values are defined: • 0: Unknown • 1: ISDN (E163/E164) • 2: Data (X.121) • 3: Telex (F.69) • 6: Land Mobile (E.212) • 8: National

	• `9`: Private • `10`: ERMES • `13`: Internet (IP) • `18`: WAP Client Id (to be defined by WAP Forum)
`CamelSmppServiceType`	The service type parameter can be used to indicate the SMS Application service associated with the message. The following generic service_types are defined: • `CMT`: Cellular Messaging • `CPT`: Cellular Paging • `VMN`: Voice Mail Notification • `VMA`: Voice Mail Alerting • `WAP`: Wireless Application Protocol • `USSD`: Unstructured Supplementary Services Data
`CamelSmppRegisteredDelivery`	Is used to request an SMSC delivery receipt and/or SME originated acknowledgements. The following values are defined: • `0`: No SMSC delivery receipt requested. • `1`: SMSC delivery receipt requested where final delivery outcome is success or failure. • `2`: SMSC delivery receipt requested where the final delivery outcome is delivery failure.
`CamelSmppPriorityFlag`	Allows the originating SME to assign a priority level to the short message. Four Priority Levels are supported: • `0`: Level 0 (lowest) priority • `1`: Level 1 priority • `2`: Level 2 priority • `3`: Level 3 (highest) priority

`CamelSmppScheduleDeliveryTime`	This parameter specifies the scheduled time at which the message delivery should be first attempted. It defines either the absolute date and time or relative time from the current SMSC time at which delivery of this message will be attempted by the SMSC. It can be specified in either absolute time format or relative time format. The encoding of a time format is specified in chapter 7.1.1. in the smpp specification v3.4.
`CamelSmppValidityPeriod`	The validity period parameter indicates the SMSC expiration time, after which the message should be discarded if not delivered to the destination. It can be defined in absolute time format or relative time format. The encoding of absolute and relative time format is specified in chapter 7.1.1 in the smpp specification v3.4.
`CamelSmppReplaceIfPresentFlag`	The replace if present flag parameter is used to request the SMSC to replace a previously submitted message, that is still pending delivery. The SMSC will replace an existing message provided that the source address, destination address and service type match the same fields in the new message. The following values are defined: • `0`: Don't replace • `1`: Replace
`CamelSmppDataCoding`	The data coding according to the SMPP 3.4 specification, section 5.2.19: `0`: SMSC Default Alphabet `4`: 8 bit Alphabet `8`: UCS2 Alphabet

The following message headers are used by the SMPP producer to set the response from the SMSC in the message header

Header	Description
`CamelSmppId`	the id to identify the submitted short message for later use (delivery receipt, query sm, cancel sm, replace sm).

The following message headers are used by the SMPP consumer to set the request data from the SMSC in the message header

Header	Description
`CamelSmppSequenceNumber`	**only for alert notification, deliver sm and data sm**: A sequence number allows a response PDU to be correlated with a request PDU. The associated SMPP response PDU must preserve this field.
`CamelSmppCommandId`	**only for alert notification, deliver sm and data sm**: The command id field identifies the particular SMPP PDU. For the complete list of defined values see chapter 5.1.2.1 in the smpp specification v3.4.

`CamelSmppSourceAddr`	**only for alert notification, deliver sm and data sm**: Defines the address of SME (Short Message Entity) which originated this message.
`CamelSmppSourceAddrNpi`	**only for alert notification and data sm**: Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. The following NPI values are defined: • `0`: Unknown • `1`: ISDN (E163/E164) • `2`: Data (X.121) • `3`: Telex (F.69) • `6`: Land Mobile (E.212) • `8`: National • `9`: Private • `10`: ERMES • `13`: Internet (IP) • `18`: WAP Client Id (to be defined by WAP Forum)
`CamelSmppSourceAddrTon`	**only for alert notification and data sm**: Defines the type of number (TON) to be used in the SME originator address parameters. The following TON values are defined: • `0`: Unknown • `1`: International • `2`: National • `3`: Network Specific • `4`: Subscriber Number • `5`: Alphanumeric • `6`: Abbreviated

`CamelSmppEsmeAddr`	**only for alert notification**: Defines the destination ESME address. For mobile terminated messages, this is the directory number of the recipient MS.
`CamelSmppEsmeAddrNpi`	**only for alert notification**: Defines the numeric plan indicator (NPI) to be used in the ESME originator address parameters. The following NPI values are defined: • `0`: Unknown • `1`: ISDN (E163/E164) • `2`: Data (X.121) • `3`: Telex (F.69) • `6`: Land Mobile (E.212) • `8`: National • `9`: Private • `10`: ERMES • `13`: Internet (IP) • `18`: WAP Client Id (to be defined by WAP Forum)
`CamelSmppEsmeAddrTon`	**only for alert notification**: Defines the type of number (TON) to be used in the ESME originator address parameters. The following TON values are defined: • `0`: Unknown • `1`: International • `2`: National • `3`: Network Specific • `4`: Subscriber Number • `5`: Alphanumeric • `6`: Abbreviated

`CamelSmppId`	**only for smsc delivery receipt and data sm**: The message ID allocated to the message by the SMSC when originally submitted.
`CamelSmppDelivered`	**only for smsc delivery receipt**: Number of short messages delivered. This is only relevant where the original message was submitted to a distribution list.The value is padded with leading zeros if necessary.
`CamelSmppDoneDate`	**only for smsc delivery receipt**: The time and date at which the short message reached it's final state. The format is as follows: YYMMDDhhmm.
`CamelSmppStatus`	**only for smsc delivery receipt and data sm**: The final status of the message. The following values are defined: • `DELIVRD`: Message is delivered to destination • `EXPIRED`: Message validity period has expired. • `DELETED`: Message has been deleted. • `UNDELIV`: Message is undeliverable • `ACCEPTD`: Message is in accepted state (i.e. has been manually read on behalf of the subscriber by customer service) • `UNKNOWN`: Message is in invalid state • `REJECTD`: Message is in a rejected state
`CamelSmppError`	**only for smsc delivery receipt**: Where appropriate this may hold a Network specific error code or an SMSC error code for the attempted delivery of the message. These errors are Network or SMSC specific and are not included here.
`CamelSmppSubmitDate`	**only for smsc delivery receipt**: The time and date at which the short message was submitted. In the case of a message which has been replaced, this is the date that the original message was replaced. The format is as follows: YYMMDDhhmm.
`CamelSmppSubmitted`	**only for smsc delivery receipt**: Number of short messages originally submitted. This is only relevant when the original message was submitted to a distribution list.The value is padded with leading zeros if necessary.
`CamelSmppDestAddr`	**only for deliver sm and data sm**: Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
`CamelSmppScheduleDeliveryTime`	**only for deliver sm and data sm**: This parameter specifies the scheduled time at which the message delivery should be first attempted.

	It defines either the absolute date and time or relative time from the current SMSC time at which delivery of this message will be attempted by the SMSC. It can be specified in either absolute time format or relative time format. The encoding of a time format is specified in Section 7.1.1. in the smpp specification v3.4.
`CamelSmppValidityPeriod`	**only for deliver sm**: The validity period parameter indicates the SMSC expiration time, after which the message should be discarded if not delivered to the destination. It can be defined in absolute time format or relative time format. The encoding of absolute and relative time format is specified in Section 7.1.1 in the smpp specification v3.4.
`CamelSmppServiceType`	**only for deliver sm and data sm**: The service type parameter indicates the SMS Application service associated with the message.
`CamelSmppRegisteredDelivery`	**only for data sm**: Is used to request an delivery receipt and/or SME originated acknowledgements. The following values are defined: `0`: No SMSC delivery receipt requested. `1`: SMSC delivery receipt requested where final delivery outcome is success or failure. `2`: SMSC delivery receipt requested where the final delivery outcome is delivery failure.
`CamelSmppDestAddrNpi`	**only for data sm**: Defines the numeric plan indicator (NPI) in the destination address parameters. The following NPI values are defined: `0`: Unknown `1`: ISDN (E163/E164) `2`: Data (X.121) `3`: Telex (F.69) `6`: Land Mobile (E.212) `8`: National `9`: Private `10`: ERMES `13`: Internet (IP) `18`: WAP Client Id (to be defined by WAP Forum)
`CamelSmppDestAddrTon`	**only for data sm**: Defines the type of number (TON) in the destination address parameters. The following TON values are defined: `0`: Unknown `1`: International `2`: National `3`: Network Specific `4`: Subscriber Number `5`: Alphanumeric `6`: Abbreviated
`CamelSmppMessageType`	**Camel 2.6 onwarts**: Identifies the type of an incoming message: `AlertNotification`: an SMSC alert notification `DataSm`: an SMSC data short message `DeliveryReceipt`: an SMSC delivery receipt `DeliverSm`: an SMSC deliver short message

JSMPP library

See the documentation of the JSMPP Library[3] for more details about the underlying library.

[3] http://code.google.com/p/jsmpp/

Samples

A route which sends an SMS using the Java DSL:

```
from("direct:start")
  .to("smpp://smppclient@localhost:2775?password=password&enquireLinkTimer=3000&transaction
Timer=5000&systemType=producer");
```

A route which sends an SMS using the Spring XML DSL:

```
<route>
  <from uri="direct:start"/>
  <to uri="smpp://smppclient@localhost:2775?password=password&nquireLinkTimer=3000&ransac
tionTimer=5000&ystemType=producer"/>
</route>
```

A route which receives an SMS using the Java DSL:

```
from("smpp://smppclient@localhost:2775?password=password&enquireLinkTimer=3000&transaction
Timer=5000&systemType=consumer")
  .to("bean:foo");
```

A route which receives an SMS using the Spring XML DSL:

```
  <route>
    <from uri="smpp://smppclient@localhost:2775?password=password&nquireLinkTimer=3000&rans
actionTimer=5000&ystemType=consumer"/>
     <to uri="bean:foo"/>
  </route>
```

SMSC simulator

If you need an SMSC simulator for your test, you can use the simulator provided by Logica[4].

Debug logging

This component has log level **DEBUG**, which can be helpful in debugging problems. If you use log4j, you can add the following line to your configuration:

```
log4j.logger.org.apache.camel.component.smpp=DEBUG
```

[4] http://opensmpp.logica.com/CommonPart/Download/download2.html#simulator

Chapter 80. SNMP

SNMP Component

The **snmp:** component gives you the ability to poll SNMP capable devices or receiving traps.

URI format

```
snmp://hostname[:port][?Options]
```

The component supports polling OID values from an SNMP enabled device and receiving traps.

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

Options

Name	Default Value	Description
`type`	none	The type of action you want to perform. Actually you can enter here `POLL` or `TRAP`. The valu to poll a given host for the supplied OID keys. If you put in `TRAP` you will setup a listener for
`address`	none	This is the IP address and the port of the host to poll or where to setup the Trap Receiver. E
`protocol`	none	Here you can select which protocol to use. By default it will be `udp` protocol but you may wa
`retries`	`2`	Defines how often a retry is made before canceling the request.
`timeout`	`1500`	Sets the timeout value for the request in millis.
`snmpVersion`	`0` (which means SNMPv1)	Sets the snmp version for the request.
`snmpCommunity`	`public`	Sets the community octet string for the snmp request.
`delay`	`60` seconds	Defines the delay in seconds between to poll cycles.
`oids`	none	Defines which values you are interested in. Please have a look at the Wikipedia[1] to get a b provide a single OID or a coma separated list of OIDs. Example: `oids="1.3.6.1.2.1.1.3.0,1.3.6.1.2.1.25.3.2.1.5.1,1.3.6.1.2.1.25.3.5.1.1.1,`

[1] http://en.wikipedia.org/wiki/Object_identifier

The result of a poll

Given the situation, that I poll for the following OIDs:

```
1.3.6.1.2.1.1.3.0
1.3.6.1.2.1.25.3.2.1.5.1
1.3.6.1.2.1.25.3.5.1.1.1
1.3.6.1.2.1.43.5.1.1.11.1
```

The result will be the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<snmp>
  <entry>
    <oid>1.3.6.1.2.1.1.3.0</oid>
    <value>6 days, 21:14:28.00</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.25.3.2.1.5.1</oid>
    <value>2</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.25.3.5.1.1.1</oid>
    <value>3</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.43.5.1.1.11.1</oid>
    <value>6</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.1.1.0</oid>
    <value>My Very Special Printer Of Brand Unknown</value>
  </entry>
</snmp>
```

As you maybe recognized there is one more result than requested....1.3.6.1.2.1.1.1.0. This one is filled in by the device automatically in this special case. So it may absolutely happen, that you receive more than you requested...be prepared.

Examples

Polling a remote device:

```
snmp:192.168.178.23:161?protocol=udp&type=POLL&oids=1.3.6.1.2.1.1.5.0
```

Setting up a trap receiver (*no OID info is needed here!*):

```
snmp:127.0.0.1:162?protocol=udp&type=TRAP
```

Routing example in Java (converts the SNMP PDU to XML String):

```
from("snmp:192.168.178.23:161?protocol=udp&type=POLL&oids=1.3.6.1.2.1.1.5.0").
convertBodyTo(String.class).
to("activemq:snmp.states");
```

Chapter 81. SpringIntegration

Spring Integration Component

The **spring-integration:** component provides a bridge for Fuse Mediation Router components to talk to spring integration endpoints[1].

URI format

```
spring-integration:defaultChannelName[?options]
```

Where **defaultChannelName** represents the default channel name which is used by the Spring Integration Spring context. It will equal to the `inputChannel` name for the Spring Integration consumer and the `outputChannel` name for the Spring Integration provider.

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

Options

Name	Description
`inputChannel`	The Spring integration input channel name that this endpoint wants to consume from, whe channel name is defined in the Spring context.
`outputChannel`	The Spring integration output channel name that is used to send messages to the Spring i
`inOut`	The exchange pattern that the Spring integration endpoint should use.
`consumer.delay`	Delay in milliseconds between each poll.
`consumer.initialDelay`	Milliseconds before polling starts.

[1] http://camel.apache.org/springintegration.html

`consumer.userFixedDelay`	Specify `true` to use fixed delay between polls, otherwise fixed rate is used. See the Java[ScheduledExecutorService|http://java.sun.com/j2se/1.5.0/docs/api/index.html?java/lang class for details.

Usage

The Spring integration component is a bridge that connects Fuse Mediation Router endpoints with Spring integration endpoints through the Spring integration's input channels and output channels. Using this component, we can send Camel messages to Spring Integration endpoints or receive messages from Spring integration endpoints in a Camel routing context.

Using the Spring integration endpoint

You can set up a Spring integration endpoint using a URI, as follows:

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:beans="http://www.springframework.org/schema/beans"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/integration
   http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
   http://camel.apache.org/schema/spring
   http://camel.apache.org/schema/spring/camel-spring.xsd">

 <channel id="inputChannel"/>
    <channel id="outputChannel"/>
    <channel id="onewayChannel"/>

 <service-activator input-channel="inputChannel"
           ref="helloService"
           method="sayHello"/>

 <service-activator input-channel="onewayChannel"
           ref="helloService"
           method="greet"/>

 <beans:bean id="helloService" class="org.apache.camel.component.spring.integration.Hello
WorldService"/>

    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
      <route>
        <from uri="direct:twowayMessage"/>
        <!-- Using the &as the separator of & -->
        <to uri="spring-integration:inputChannel?inOut=true&nputChannel=outputChannel"/>
      </route>
```

```
      <route>
        <from uri="direct:onewayMessage"/>
        <to uri="spring-integration:onewayChannel?inOut=false"/>
      </route>
    </camelContext>
```

```
<channel id="requestChannel"/>
<channel id="responseChannel"/>

<beans:bean id="myProcessor" class="org.apache.camel.component.spring.integration.MyPro
cessor"/>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <!-- Using the &as the separator of & -->
   <from uri="spring-integration://requestChannel?outputChannel=responseChannel&nOut=true"/>

    <process ref="myProcessor"/>
  </route>
</camelContext>
```

Or directly using a Spring integration channel name:

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:beans="http://www.springframework.org/schema/beans"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/integration
   http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
   http://camel.apache.org/schema/spring
   http://camel.apache.org/schema/spring/camel-spring.xsd">
 <channel id="outputChannel"/>

    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
      <route>
        <!-- camel will create a spring integration endpoint automatically -->
        <from uri="outputChannel"/>
        <to uri="mock:result"/>
      </route>
    </camelContext>
```

The Source and Target adapter

Spring integration also provides the Spring integration's source and target adapters, which can route messages from a Spring integration channel to a Fuse Mediation Router endpoint or from a Fuse Mediation Router endpoint to a Spring integration channel.

This example uses the following namespaces:

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
       xmlns:beans="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel-si="http://camel.apache.org/schema/spring/integration"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
       http://camel.apache.org/schema/spring/integration
       http://camel.apache.org/schema/spring/integration/camel-spring-integration.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd
    ">
```

You can bind your source or target to a Fuse Mediation Router endpoint as follows:

```
<!-- Create the camel context here -->
<camelContext id="camelTargetContext" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:EndpointA" />
        <to uri="mock:result" />
    </route>
    <route>
        <from uri="direct:EndpointC"/>
        <process ref="myProcessor"/>
      </route>
</camelContext>

<!-- We can bind the camelTarget to the camel context's endpoint by specifying the
camelEndpointUri attribute -->
<camel-si:camelTarget id="camelTargetA" camelEndpointUri="direct:EndpointA" ex
pectReply="false">
    <camel-si:camelContextRef>camelTargetContext</camel-si:camelContextRef>
</camel-si:camelTarget>

<camel-si:camelTarget id="camelTargetB" camelEndpointUri="direct:EndpointC" replyChannel="chan
nelC" expectReply="true">
    <camel-si:camelContextRef>camelTargetContext</camel-si:camelContextRef>
</camel-si:camelTarget>

<camel-si:camelTarget id="camelTargetD" camelEndpointUri="direct:EndpointC" ex
pectReply="true">
    <camel-si:camelContextRef>camelTargetContext</camel-si:camelContextRef>
</camel-si:camelTarget>
```

```
<beans:bean id="myProcessor" class="org.apache.camel.component.spring.integration.MyPro
cessor"/>
```

```
<!-- spring integration channels -->
<channel id="channelA"/>
<channel id="channelB"/>
<channel id="channelC"/>

<!-- spring integration service activator -->
<service-activator input-channel="channelB" output-channel="channelC" ref="helloService"
method="sayHello"/>

<!-- custom bean -->
<beans:bean id="helloService" class="org.apache.camel.component.spring.integration.Hello
WorldService"/>

<camelContext id="camelSourceContext" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:OneWay"/>
        <to uri="direct:EndpointB" />
    </route>
    <route>
        <from uri="direct:TwoWay"/>
        <to uri="direct:EndpointC" />
    </route>
</camelContext>

<!-- camelSource will redirect the message coming for direct:EndpointB to the spring request
Channel channelA -->

<camel-si:camelSource id="camelSourceA" camelEndpointUri="direct:EndpointB"
                        requestChannel="channelA" expectReply="false">
    <camel-si:camelContextRef>camelSourceContext</camel-si:camelContextRef>
</camel-si:camelSource>

<!-- camelSource will redirect the message coming for direct:EndpointC to the spring request
Channel channelB
 then it will pull the response from channelC and put the response message back to direct:En
dpointC -->

<camel-si:camelSource id="camelSourceB" camelEndpointUri="direct:EndpointC"
    requestChannel="channelB" replyChannel="channelC" expectReply="true">
    <camel-si:camelContextRef>camelSourceContext</camel-si:camelContextRef>
</camel-si:camelSource>
```

Chapter 82. Spring Security

Spring Security

Available as of Camel 2.3

The **camel-spring-security** component provides role-based authorization for Camel routes. It leverages the authentication and user services provided by Spring Security[1] (formerly Acegi Security) and adds a declarative, role-based policy system to control whether a route can be executed by a given principal.

If you are not familiar with the Spring Security authentication and authorization system, please review the current reference documentation on the SpringSource web site linked above.

Creating authorization policies

Access to a route is controlled by an instance of a `SpringSecurityAuthorizationPolicy` object. A policy object contains the name of the Spring Security authority (role) required to run a set of endpoints and references to Spring Security `AuthenticationManager` and `AccessDecisionManager` objects used to determine whether the current principal has been assigned that role. Policy objects may be configured as Spring beans or by using an `<authorizationPolicy>` element in Spring XML.

The `<authorizationPolicy>` element may contain the following attributes:

Name	Default Value	Description
`id`	`null`	The unique Spring bean identifier which is used to reference the policy in routes (required)
`access`	`null`	The Spring Security authority name that is passed to the access decision manager (required)
`authenticationManager`	`authenticationManager`	The name of the Spring Security `AuthenticationManager` object in the context
`accessDecisionManager`	`accessDecisionManager`	The name of the Spring Security `AccessDecisionManager` object in the context
`authenticationAdapter`	DefaultAuthenticationAdapter	**Camel 2.4** The name of a **camel-spring-security**`AuthenticationAdapter` object in the context that is used to convert a `javax.security.auth.Subject` into a Spring Security `Authentication` instance.

[1] http://static.springsource.org/spring-security/site/index.html

useThreadSecurityContext	true	If a `javax.security.auth.Subject` cannot be found in the In message header under Exchange.AUTHENTICATION, check the Spring Security `SecurityContextHolder` for an `Authentication` object.
alwaysReauthenticate	false	If set to true, the `SpringSecurityAuthorizationPolicy` will always call `AuthenticationManager.authenticate()` each time the policy is accessed.

Controlling access to Camel routes

A Spring Security `AuthenticationManager` and `AccessDecisionManager` are required to use this component. Here is an example of how to configure these objects in Spring XML using the Spring Security namespace:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:spring-security="http://www.springframework.org/schema/security"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/security
   http://www.springframework.org/schema/security/spring-security.xsd">

    <bean id="accessDecisionManager" class="org.springframework.security.access.vote.Affirm
ativeBased">
        <property name="allowIfAllAbstainDecisions" value="true"/>
        <property name="decisionVoters">
            <list>
                <bean class="org.springframework.security.access.vote.RoleVoter"/>
            </list>
        </property>
    </bean>

    <spring-security:authentication-manager alias="authenticationManager">
     <spring-security:authentication-provider user-service-ref="userDetailsService"/>
    </spring-security:authentication-manager>

    <spring-security:user-service id="userDetailsService">
        <spring-security:user name="jim" password="jimspassword" authorities="ROLE_USER,
ROLE_ADMIN"/>
        <spring-security:user name="bob" password="bobspassword" authorities="ROLE_USER"/>

    </spring-security:user-service>

</beans>
```

Now that the underlying security objects are set up, we can use them to configure an authorization policy and use that policy to control access to a route:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:spring-security="http://www.springframework.org/schema/security"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
          http://camel.apache.org/schema/spring
          http://camel.apache.org/schema/spring/camel-spring.xsd
          http://camel.apache.org/schema/spring-security
          http://camel.apache.org/schema/spring-security/camel-spring-security.xsd
          http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.0.3.xsd">

    <!-- import the spring security configuration  -->
    <import resource="classpath:org/apache/camel/component/spring/security/commonSecur
ity.xml"/>

    <authorizationPolicy id="admin" access="ROLE_ADMIN"
                         authenticationManager="authenticationManager"
                         accessDecisionManager="accessDecisionManager"
                         xmlns="http://camel.apache.org/schema/spring-security"/>

    <camelContext id="myCamelContext" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="direct:start"/>
            <!-- The exchange should be authenticated with the role of ADMIN before it is
send to mock:endpoint -->
            <policy ref="admin">
                <to uri="mock:end"/>
            </policy>
        </route>
    </camelContext>

</beans>
```

In this example, the endpoint `mock:end` will not be executed unless a Spring Security `Authentication` object that has been or can be authenticated and contains the `ROLE_ADMIN` authority can be located by the *admin*`SpringSecurityAuthorizationPolicy`.

Authentication

The process of obtaining security credentials that are used for authorization is not specified by this component. You can write your own processors or components which get authentication information from the exchange depending on your needs. For example, you might create a processor that gets credentials from an HTTP request header originating in the camel-jetty component. No matter how the credentials are collected, they

need to be placed in the In message or the `SecurityContextHolder` so the **camel-spring-security** component can access them:

```
import javax.security.auth.Subject;
import org.apache.camel.*;
import org.apache.commons.codec.binary.Base64;
import org.springframework.security.authentication.*;


public class MyAuthService implements Processor {
    public void process(Exchange exchange) throws Exception {
        // get the username and password from the HTTP header
        // http://en.wikipedia.org/wiki/Basic_access_authentication
       String userpass = new String(Base64.decodeBase64(exchange.getIn().getHeader("Author
ization", String.class)));
        String[] tokens= userpass.split(":");

        // create an Authentication object
        UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthentication
Token(tokens[0], tokens[1]);

        // wrap it in a Subject
        Subject subject = new Subject();
        subject.getPrincipals().add(token);

        // place the Subject in the In message
        exchange.getIn().setHeader(Exchange.AUTHENTICATION, subject);

        // you could also do this if useThreadSecurityContext is set to true
        // SecurityContextHolder.getContext().setAuthentication(authToken);
    }
}
```

The `SpringSecurityAuthorizationPolicy` will automatically authenticate the `Authentication` object if necessary.

There are two issues to be aware of when using the `SecurityContextHolder` instead of or in addition to the `Exchange.AUTHENTICATION` header. First, the context holder uses a thread-local variable to hold the `Authentication` object. Any routes that cross thread boundaries, like **seda** or **jms**, will lose the `Authentication` object. Second, the Spring Security system appears to expect that an `Authentication` object in the context is already authenticated and has roles (see the Technical Overview section 5.3.1[2] for more details).

The default behavior of **camel-spring-security** is to look for a `Subject` in the `Exchange.AUTHENTICATION` header. This `Subject` must contain at least one principal, which must be a subclass of `org.springframework.security.core.Authentication`. You can customize the mapping of `Subject` to `Authentication` object by providing an implementation of the

[2] http://static.springsource.org/spring-security/site/docs/3.0.x/reference/technical-overview.html#tech-intro-authentication

`org.apache.camel.component.spring.security.AuthenticationAdapter` to your `<authorizationPolicy>` bean. This can be useful if you are working with components that do not use Spring Security but do provide a `Subject`. At this time, only the camel-cxf component populates the `Exchange.AUTHENTICATION` header.

Handling authentication and authorization errors

If authentication or authorization fails in the `SpringSecurityAuthorizationPolicy`, a `CamelAuthorizationException` will be thrown. This can be handled using Camel's standard exception handling methods, like the Exception clause[3]. The `CamelAuthorizationException` will have a reference to the ID of the policy which threw the exception so you can handle errors based on the policy as well as the type of exception:

```
<onException>
  <exception>org.springframework.security.authentication.AccessDeniedException</exception>

  <choice>
    <when>
      <simple>${exception.policyId} == 'user'</simple>
      <transform>
        <constant>You do not have ROLE_USER access!</constant>
      </transform>
    </when>
    <when>
      <simple>${exception.policyId} == 'admin'</simple>
      <transform>
        <constant>You do not have ROLE_ADMIN access!</constant>
      </transform>
    </when>
  </choice>
</onException>
```

Dependencies

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

```
 <dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-security</artifactId>
  <version>2.4.0</version>
</dependency>
```

This dependency will also pull in `org.springframework.security:spring-security-core:3.0.3.RELEASE` and `org.springframework.security:spring-security-config:3.0.3.RELEASE`.

[3] exception-clause

Chapter 83. Spring Web Services

Spring Web Services Component

The **spring-ws:** component allows you to integrate with Spring Web Services[1]. It offers both *clientside support, for accessing web services, and* serverside support for creating your own contract-first web services.

Dependencies

This component offers support for Spring-WS 1.5.9 which is compatible with Spring 2.5.x and 3.0.x. In order to run camel-spring-ws on Spring 2.5.x you need to add the spring-webmvc module from Spring 2.5.x.

In order to run Spring-WS 1.5.9 on Spring 3.0 you need to exclude the OXM module from Spring 3.0 as this module is also included in Spring-WS 1.5.9.

URI format

The URI scheme for this component is as follows

```
spring-ws:[mapping-type:]address[?options]
```

To expose a web service, `mapping-type` needs to be set to one of the following values:

Mapping type	Description
`rootqname`	Offers the option to map web service requests based on the qualified name of the root element contained in the message.
`soapaction`	Used to map web service requests based on the SOAP action specified in the header of the message.
`uri`	In order to map web service requests that target a specific URI.
`xpathresult`	Used to map web service requests based on the evaluation of an XPath `expression` against the incoming message. The result of the evaluation should match the XPath result specified in the endpoint URI.
`beanname`	Allows you to reference a `org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher` in order to

[1] http://static.springsource.org/spring-ws/sites/1.5/

	integrate with existing (legacy) endpoint mappings[2] like `PayloadRootQNameEndpointMapping`, `SoapActionEndpointMapping`, etc

As a consumer the `address` should contain a value relevant to the specified mapping-type (e.g. a SOAP action, XPath expression). As a producer the address should be set to the URI of the web service you are calling upon.

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

Options

Name	Required?	Description
`soapAction`	*No*	SOAP action to include inside a SOAP request when accessing remote web services
`wsAddressingAction`	*No*	WS-Addressing 1.0 action header to include when accessing web services. The `To` header is set to the *address* of the web service as specified in the endpoint URI (default Spring-WS behavior).
`expression`	Only when *mapping-type* is `xpathresult`	XPath expression to use in the process of mapping web service requests, should match the result specified by `xpathresult`

Registry based options

The following options can be specified in the registry (most likely a Spring application context) and referenced from the endpoint URI using the `#beanID` notation.

Name	Required?	Description
`webServiceTemplate`	*No*	Option to provide a custom WebServiceTemplate[3]. This allows for full control over client-side web services handling; like adding a custom interceptor or specifying a fault resolver, message sender or message factory.
`messageSender`	*No*	Option to provide a custom WebServiceMessageSender[4]. For example to perform authentication or use alternative transports
`messageFactory`	*No*	Option to provide a custom WebServiceMessageFactory[5]. For example when you want Apache Axiom to handle web service messages instead of SAAJ

[2] http://static.springsource.org/spring-ws/sites/1.5/reference/html/server.html#server-endpoint-mapping
[3] http://static.springsource.org/spring-ws/sites/1.5/apidocs/org/springframework/ws/client/core/WebServiceTemplate.html
[4] http://static.springsource.org/spring-ws/sites/1.5/apidocs/org/springframework/ws/transport/WebServiceMessageSender.html
[5] http://static.springsource.org/spring-ws/sites/1.5/apidocs/org/springframework/ws/WebServiceMessageFactory.html

`transformerFactory`	*No*	Option to override the default `TransformerFactory`. The provided transformer factory must be of type `javax.xml.transform.TransformerFactory`
`endpointMapping`	Only when *mapping-type* is `rootqname`, `soapaction`, `uri` or `xpathresult`	Reference to `org.apache.camel.component.spring.ws.bean.CamelEndpointMapping` in the Registry/ApplicationContext. Only one bean is required in the registry to serve all Camel/Spring-WS endpoints. This bean is auto-discovered by the MessageDispatcher[6] and used to map requests to Camel endpoints based on characteristics specified on the endpoint (like root QName, SOAP action, etc)

Message headers

Name	Type	Description
`CamelSpringWebserviceEndpointUri`	`String`	URI of the web service you are accessing as a client; overrides the *address* part of the endpoint URI.
`CamelSpringWebserviceSoapAction`	`String`	Header to specify the SOAP action of the message; overrides the `soapAction` option, if present
`CamelSpringWebserviceAddressingAction`	`URI`	Use this header to specify the WS-Addressing action of the message; overrides the `wsAddressingAction` option, if present

Accessing web services

To call a web service at `http://foo.com/bar` simply define a route:

```
from("direct:example").to("spring-ws:http://foo.com/bar")
```

And sent a message:

```
template.requestBody("direct:example", "<foobar xmlns=\"http://foo.com\"><msg>test mes
sage</msg></foobar>");
```

Sending SOAP and WS-Addressing action headers

When a remote web service requires a SOAP action or use of the WS-Addressing standard you define your route as:

[6] http://static.springsource.org/spring-ws/sites/1.5/apidocs/org/springframework/ws/server/MessageDispatcher.html

```
from("direct:example")
.to("spring-ws:http://foo.com/bar?soapAction=http://foo.com&wsAddressingAction=ht
tp://bar.com")
```

Optionally you can override the endpoint options with header values:

```
template.requestBodyAndHeader("direct:example",
"<foobar xmlns=\"http://foo.com\"><msg>test message</msg></foobar>",
SpringWebserviceConstants.SPRING_WS_SOAP_ACTION, "http://baz.com");
```

Using a custom MessageSender and MessageFactory

A custom message sender or factory in the registry can be referenced like this:

```
from("direct:example")
.to("spring-ws:http://foo.com/bar?messageFactory=#messageFactory&messageSender=#mes
sageSender")
```

Spring configuration:

```
<!-- authenticate using HTTP Basic Authentication -->
<bean id="messageSender" class="org.springframework.ws.transport.http.CommonsHttpMes
sageSender">
 <property name="credentials">
  <bean class="org.apache.commons.httpclient.UsernamePasswordCredentials">
   <constructor-arg index="0" value="admin"/>
   <constructor-arg index="1" value="secret"/>
  </bean>
 </property>
</bean>

<!-- force use of Sun SAAJ implementation, http://static.springsource.org/spring-
ws/sites/1.5/faq.html#saaj-jboss -->
<bean id="messageFactory" class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory">

 <property name="messageFactory">
  <bean class="com.sun.xml.messaging.saaj.soap.ver1_1.SOAPMessageFactory1_1Impl"></bean>
 </property>
</bean>
```

Exposing web services

In order to expose a web service using this component you first need to set-up a MessageDispatcher[7] to look for endpoint mappings in a Spring XML file. If you plan on running inside a servlet container you probably want to use a `MessageDispatcherServlet` configured in `web.xml`.

[7] http://static.springsource.org/spring-ws/sites/1.5/reference/html/server.html

By default the `MessageDispatcherServlet` will look for a Spring XML named `/WEB-INF/spring-ws-servlet.xml`. To use Camel with Spring-WS the only mandatory bean in that XML file is `CamelEndpointMapping`. This bean allows the `MessageDispatcher` to dispatch web service requests to your routes.

web.xml

```
<web-app>
    <servlet>
        <servlet-name>spring-ws</servlet-name>
        <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</ser
vlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

spring-ws-servlet.xml

```
<bean id="endpointMapping" class="org.apache.camel.component.spring.ws.bean.CamelEndpoint
Mapping" />

<bean id="wsdl" class="org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition">
   <property name="schema">
      <bean class="org.springframework.xml.xsd.SimpleXsdSchema">
         <property name="xsd" value="/WEB-INF/foobar.xsd"/>
      </bean>
   </property>
   <property name="portTypeName" value="FooBar"/>
   <property name="locationUri" value="/"/>
   <property name="targetNamespace" value="http://example.com/"/>
</bean>
```

More information on setting up Spring-WS can be found in Writing Contract-First Web Services[8].

Endpoint mapping in routes

With the XML configuration in-place you can now use Camel's DSL to define what web service requests are handled by your endpoint. The following route will receive all web service requests that have a root element named `GetFoo` within the `http://example.com/` namespace:

```
from("spring-ws:rootqname:{http://example.com/}GetFoo?endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

[8] http://static.springsource.org/spring-ws/sites/1.5/reference/html/tutorial.html

The following route will receive web service requests containing the `http://example.com/GetFoo` SOAP action:

```
from("spring-ws:soapaction:http://example.com/GetFoo?endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

The following route will receive all requests sent to `http://example.com/foobar`:

```
from("spring-ws:uri:http://example.com/foobar?endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

The route below receives requests that contain the element `<foobar>abc</foobar>` anywhere inside the message (and the default namespace).

```
from("spring-ws:xpathresult:abc?expression=//foobar&endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

Alternative configuration, using existing endpoint mappings

For every endpoint with mapping-type `beanname` one bean of type `CamelEndpointDispatcher` with a corresponding name is required in the Registry/ApplicationContext. This bean acts as a bridge between the Camel endpoint and an existing endpoint mapping[9] like `PayloadRootQNameEndpointMapping`.

Note

The use of the `beanname` mapping-type is primarily meant for (legacy) situations where you're already using Spring-WS and have endpoint mappings defined in a Spring XML file. The `beanname` mapping-type allows you to wire your Camel route into an existing endpoint mapping. When you're starting from scratch it's recommended to define your endpoint mappings as Camel URI's (as illustrated above with `endpointMapping`) since it requires less configuration and is more expressive. Alternatively you could use vanilla Spring-WS with the help of annotations.

An example of a route using `beanname`:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
 <route>
  <from uri="spring-ws:beanname:QuoteEndpointDispatcher" />
  <to uri="mock:example" />
 </route>
</camelContext>

<bean id="legacyEndpointMapping" class="org.springframework.ws.server.endpoint.mapping.Pay
```

[9] http://static.springsource.org/spring-ws/sites/1.5/reference/html/server.html#server-endpoint-mapping

```
loadRootQNameEndpointMapping">
    <property name="mappings">
        <props>
            <prop key="{http://example.com/}GetFuture">FutureEndpointDispatcher</prop>
            <prop key="{http://example.com/}GetQuote">QuoteEndpointDispatcher</prop>
        </props>
    </property>
</bean>

<bean id="QuoteEndpointDispatcher" class="org.apache.camel.component.spring.ws.bean.CamelEnd
pointDispatcher" />
<bean id="FutureEndpointDispatcher" class="org.apache.camel.component.spring.ws.bean.CamelEnd
pointDispatcher" />
```

POJO (un)marshalling

Camel's pluggable data formats offer support for POJO/XML marshalling using libraries such as JAXB, XStream, Castor and XMLBeans. You can use these data formats in your route to sent and receive POJOs (Plain Old Java Objects), to and from web services.

When *accessing* web services you can marshal the request and unmarshal the response message:

```
JaxbDataFormat jaxb = new JaxbDataFormat(false);
jaxb.setContextPath("com.example.model");

from("direct:example").marshal(jaxb).to("spring-ws:http://foo.com/bar").unmarshal(jaxb);
```

Similarly when *providing* web services, you can unmarshal XML requests to POJOs and marshal the response message back to XML:

```
from("spring-ws:rootqname:{http://example.com/}GetFoo?endpointMapping=#endpointMapping").un
marshal(jaxb)
.to("mock:example").marshal(jaxb);
```

Chapter 84. SQL Component

SQL Component

The **sql:** component allows you to work with databases using JDBC queries. The difference between this component and JDBC on page 267 component is that in case of SQL the query is a property of the endpoint and it uses message payload as parameters passed to the query.

This component uses `spring-jdbc` behind the scenes for the actual SQL handling.

URI format

Warning

The SQL component can only be used to define producer endpoints. In other words, you cannot define an SQL endpoint in a `from()` statement.

The SQL component uses the following endpoint URI notation:

```
sql:select * from table where id=# order by name[?options]
```

Notice that the standard `?` symbol that denotes the parameters to an SQL query is substituted with the `#` symbol, because the `?` symbol is used to specify options for the endpoint. The `?` symbol replacement can be configured on endpoint basis.

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

Options

Option	Type	Default	Description
`dataSourceRef`	`String`	`null`	**Fuse Mediation Router 1.5.1/2.0:** Reference to a `DataSource` to look up in the registry.
`placeholder`	`String`	`#`	**Camel 2.4:** Specifies a character that will be replaced to `?` in SQL query. Notice, that it is simple `String.replaceAll()` operation and no SQL parsing is involved (quoted strings will also change)
`template.<xxx>`		`null`	Sets additional options on the Spring `JdbcTemplate` that is used behind the scenes to execute the queries. For instance, `template.maxRows=10`.

	For detailed documentation, see the JdbcTemplate javadoc[1] documentation.

Treatment of the message body

The SQL component tries to convert the message body to an object of `java.util.Iterator` type and then uses this iterator to fill the query parameters (where each query parameter is represented by a `#` symbol, or other configured placeholder, in the endpoint URI). If the message body is not an array or collection, the conversion results in an iterator that iterates over only one object, which is the body itself.

For example, if the message body is an instance of `java.util.List`, the first item in the list is substituted into the first occurrence of `#` in the SQL query, the second item in the list is substituted into the second occurrence of `#`, and so on.

Result of the query

For `select` operations, the result is an instance of `List<Map<String, Object>>` type, as returned by the JdbcTemplate.queryForList()[2] method. For `update` operations, the result is the number of updated rows, returned as an `Integer`.

Header values

When performing `update` operations, the SQL Component stores the update count in the following message headers:

Header	Description
`SqlProducer.UPDATE_COUNT`	Fuse Mediation Router 1.x: The number of rows updated for `update` operations, returned as an `Integer` object.
`CamelSqlUpdateCount`	Fuse Mediation Router 2.0: The number of rows updated for `update` operations, returned as an `Integer` object.
`CamelSqlRowCount`	Fuse Mediation Router 2.0: The number of rows returned for `select` operations, returned as an `Integer` object.

Configuration in Fuse Mediation Router 1.5.0 or lower

The SQL component must be configured before it can be used. In Spring, you can configure it as follows:

[1] http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/jdbc/core/JdbcTemplate.html

[2] http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/jdbc/core/JdbcTemplate.html#queryForList(java.lang.String,%20java.lang.Object%91%93)

```
<bean id="sql" class="org.apache.camel.component.sql.SqlComponent">
    <property name="dataSource" ref="myDS"/>
</bean>

<bean id="myDS" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/ds" />
    <property name="username" value="username" />
    <property name="password" value="password" />
</bean>
```

Configuration in Fuse Mediation Router 1.5.1 or higher

You can now set a reference to a `DataSource` in the URI directly:

```
sql:select * from table where id=# order by name?dataSourceRef=myDS
```

Sample

In the sample below we execute a query and retrieve the result as a `List` of rows, where each row is a `Map<String, Object` and the key is the column name.

First, we set up a table to use for our sample. As this is based on an unit test, we do it java code:

```
// this is the database we create with some initial data for our unit test
jdbcTemplate.execute("create table projects (id integer primary key,"
                     + "project varchar(10), license varchar(5))");
jdbcTemplate.execute("insert into projects values (1, 'Camel', 'ASF')");
jdbcTemplate.execute("insert into projects values (2, 'AMQ', 'ASF')");
jdbcTemplate.execute("insert into projects values (3, 'Linux', 'XXX')");
```

Then we configure our route and our `sql` component. Notice that we use a `direct` endpoint in front of the `sql` endpoint. This allows us to send an exchange to the `direct` endpoint with the URI, `direct:simple`, which is much easier for the client to use than the long `sql:` URI. Note that the `DataSource` is looked up up in the registry, so we can use standard Spring XML to configure our `DataSource`.

```
from("direct:simple")
    .to("sql:select * from projects where license = # order by id?dataSourceRef=jdbc/myData
Source")
    .to("mock:result");
```

And then we fire the message into the `direct` endpoint that will route it to our `sql` component that queries the database.

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedMessageCount(1);
```

```
// send the query to direct that will route it to the sql where we will execute the query
// and bind the parameters with the data from the body. The body only contains one value
// in this case (XXX) but if we should use multi values then the body will be iterated
// so we could supply a List<String> instead containing each binding value.
template.sendBody("direct:simple", "XXX");

mock.assertIsSatisfied();

// the result is a List
List received = assertIsInstanceOf(List.class, mock.getReceivedExchanges().get(0).getIn().get
Body());

// and each row in the list is a Map
Map row = assertIsInstanceOf(Map.class, received.get(0));

// and we should be able the get the project from the map that should be Linux
assertEquals("Linux", row.get("PROJECT"));
```

We could configure the `DataSource` in Spring XML as follows:

```
 <jee:jndi-lookup id="myDS" jndi-name="jdbc/myDataSource"/>
```

See also:

- JDBC on page 267

Chapter 85. Stream

Stream Component

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

URI format

```
stream:in[?options]
stream:out[?options]
stream:err[?options]
stream:header[?options]
```

In addition, the `file` and `url` endpoint URIs are supported in **Fuse Mediation Router 2.0**:

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

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

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

Options

Name	Default Value	Description
`delay`	`0`	Initial delay in milliseconds before consuming or producing the stream.
`encoding`	*JVM Default*	As of 1.4, you can configure the encoding (is a charset name[1]) to use text-based streams (for example, message body is a `String` object). If not provided, Fuse Mediation Router uses the JVM default Charset[2].
`promptMessage`	`null`	**Fuse Mediation Router 2.0:** Message prompt to use when reading from `stream:in`; for example, you could set this to `Enter a command:`
`promptDelay`	`0`	**Fuse Mediation Router 2.0:** Optional delay in milliseconds before showing the message prompt.
`initialPromptDelay`	`2000`	**Fuse Mediation Router 2.0:** Initial delay in milliseconds before showing the message prompt. This delay occurs only once. Can be used during

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/nio/charset/Charset.html
[2] http://java.sun.com/j2se/1.5.0/docs/api/java/nio/charset/Charset.html#defaultCharset()

		system startup to avoid message prompts being written while other logging is done to the system out.
`fileName`	`null`	**Fuse Mediation Router 2.0:** When using the `stream:file` URI format, this option specifies the filename to stream to/from.
`scanStream`	`false`	**Fuse Mediation Router 2.0:** To be used for continuously reading a stream such as the unix `tail` command. **Fuse Mediation Router 2.4:** will retry opening the file if it is overwritten, like `tail --retry`
`scanStreamDelay`	`0`	**Fuse Mediation Router 2.0:** Delay in milliseconds between read attempts when using `scanStream`.
`groupLines`	`0`	**Camel 2.5:** To group X number of lines in the consumer. For example to group 10 lines and therefore only spit out an Exchange with 10 lines, instead of 1 Exchange per line.

Message content

The **stream:** component supports either `String` or `byte[]` for writing to streams. Just add either `String` or `byte[]` content to the `message.in.body`. The special `stream:header` URI is used for custom output streams. Just add a `java.io.OutputStream` object to `message.in.header` in the key `header`. See samples for an example.

Samples

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

```
@Test
public void testStringContent() throws Exception {
    template.sendBody("direct:in", "Hello Text World\n");
}

@Test
public void testBinaryContent() {
    template.sendBody("direct:in", "Hello Bytes World\n".getBytes());
}

protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {
            from("direct:in").to("stream:out");
        }
    };
}
```

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

```
private OutputStream mystream = new MyOutputStream();
private StringBuffer sb = new StringBuffer();

@Test
public void testStringContent() {
    template.sendBody("direct:in", "Hello");
    // StreamProducer appends \n in text mode
    assertEquals("Hello\n", sb.toString());
}

@Test
public void testBinaryContent() {
    template.sendBody("direct:in", "Hello".getBytes());
    // StreamProducer is in binary mode so no \n is appended
    assertEquals("Hello", sb.toString());
}

protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {
            from("direct:in").setHeader("stream", constant(mystream)).
                to("stream:header");
        }
    };
}

private class MyOutputStream extends OutputStream {

    public void write(int b) throws IOException {
        sb.append((char)b);
    }
}
```

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

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

Chapter 86. StringTemplate

String Template

The **string-template:** component allows you to process a message using a String Template[1]. This can be ideal when using Templating to generate responses for requests.

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

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

URI format

```
string-template:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template.

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

Options

Option	Default	Description
`contentCache`	`false`	New option in Fuse Mediation Router 1.4. Cache for the resource content when its loaded.

Headers

Fuse Mediation Router will store a reference to the resource in the message header with key, `org.apache.camel.stringtemplate.resource`. The Resource is an `org.springframework.core.io.Resource` object.

[1] http://www.stringtemplate.org/

Hot reloading

The string template resource is by default hot-reloadable for both file and classpath resources (expanded jar). If you set `contentCache=true`, Fuse Mediation Router loads the resource only once and hot-reloading is not possible. This scenario can be used in production when the resource never changes.

StringTemplate Attributes

Fuse Mediation Router will provide exchange information as attributes (just a `java.util.Map`) to the string template. The Exchange is transfered as:

key	value
`exchange`	The Exchange itself.
`headers`	The headers of the In message.
`camelContext`	The Camel Context.
`request`	The In message.
`in`	The In message.
`body`	The In message body.
`out`	The Out message (only for InOut message exchange pattern).
`response`	The Out message (only for InOut message exchange pattern).

Samples

For example you could use a string template as follows in order to formulate a response to a message:

```
from("activemq:My.Queue").
  to("string-template:com/acme/MyResponse.tm");
```

The Email Sample

In this sample we want to use a string template to send an order confirmation email. The email template is laid out in `StringTemplate` as:

```
Dear $headers.lastName$, $headers.firstName$

Thanks for the order of $headers.item$.

Regards Camel Riders Bookstore
$body$
```

And the java code is as follows:

```
private Exchange createLetter() {
    Exchange exchange = context.getEndpoint("direct:a").createExchange();
    Message msg = exchange.getIn();
    msg.setHeader("firstName", "Claus");
    msg.setHeader("lastName", "Ibsen");
    msg.setHeader("item", "Camel in Action");
    msg.setBody("PS: Next beer is on me, James");
    return exchange;
}

@Test
public void testVelocityLetter() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
   mock.expectedBodiesReceived("Dear Ibsen, Claus! Thanks for the order of Camel in Action.
 Regards Camel Riders Bookstore PS: Next beer is on me, James");

    template.send("direct:a", createLetter());

    mock.assertIsSatisfied();
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
          from("direct:a").to("string-template:org/apache/camel/component/stringtemplate/let
ter.tm").to("mock:result");
        }
    };
}
```

Chapter 87. Test

Test Component

The **test** component extends the Mock on page 357 component to support pulling messages from another endpoint on startup to set the expected message bodies on the underlying Mock on page 357 endpoint. That is, you use the test endpoint in a route and messages arriving on it will be implicitly compared to some expected messages extracted from some other location.

So you can use, for example, an expected set of message bodies as files. This will then set up a properly configured Mock on page 357 endpoint, which is only valid if the received messages match the number of expected messages and their message payloads are equal.

URI format

```
test:expectedMessagesEndpointUri
```

Where **expectedMessagesEndpointUri** refers to some other Component URI that the expected message bodies are pulled from before starting the test.

Example

For example, you could write a test case as follows:

```
from("seda:someEndpoint").
  to("test:file://data/expectedOutput?noop=true");
```

If your test then invokes the MockEndpoint.assertIsSatisfied(camelContext) method[1], your test case will perform the necessary assertions.

Here is a real example test case using Mock and Spring[2] along with its Spring XML[3].

To see how you can set other expectations on the test endpoint, see the Mock on page 357 component.

---

1 http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html#assertIsSatisfied(org.apache.camel.CamelContext)

2 http://svn.apache.org/viewvc/camel/trunk/components/camel-spring/src/test/java/org/apache/camel/component/test/TestEndpointTest.java?view=markup

3 https://svn.apache.org/repos/asf/camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/component/test/TestEndpointTest-context.xml

Chapter 88. Timer

Timer Component

The **timer:** component is used to generate message exchanges when a timer fires You can only consume events from this endpoint.

URI format

```
timer:name[?options]
```

Where `name` is the name of the `Timer` object, which is created and shared across endpoints. So if you use the same name for all your timer endpoints, only one `Timer` object and thread will be used.

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

Note: The IN body of the generated exchange is `null`. So `exchange.getIn().getBody()` returns `null`.

Advanced Scheduler

See also the Quartz on page 389 component that supports much more advanced scheduling.

Specify time in human friendly format

In **Camel 2.3** onwards you can specify the time in human friendly syntax[1].

Options

Name	Default Value	Description
`time`	`null`	A `java.util.Date` the **first** event should be generated. If using the URI, the pattern expected is: `yyyy-MM-dd HH:mm:ss` or `yyyy-MM-dd'T'HH:mm:ss`.
`pattern`	`null`	**Fuse Mediation Router 1.6.2/2.0:** Allows you to specify a custom `Date` pattern to use for setting the time option using URI syntax.
`period`	`1000`	If greater than 0, generate periodic events every `period` milliseconds.

[1] How do I specify time period in a human friendly syntax

`delay`	`0`	The number of milliseconds to wait before the first event is generated. Should not be used in conjunction with the `time` option.
`fixedRate`	`false`	Events take place at approximately regular intervals, separated by the specified period.
`daemon`	`true`	Specifies whether or not the thread associated with the timer endpoint runs as a daemon.

Exchange Properties

When the timer is fired, it adds the following information as properties to the `Exchange`:

Name	Type	Description
`org.apache.camel.timer.name`	`String`	The value of the `name` option.
`org.apache.camel.timer.time`	`Date`	The value of the `time` option.
`org.apache.camel.timer.period`	`long`	The value of the `period` option.
`org.apache.camel.timer.firedTime`	`Date`	**Fuse Mediation Router 1.5**: The time when the consumer fired.

Message Headers

When the timer is fired, it adds the following information as headers to the IN message

Name	Type	Description
`firedTime`	`java.util.Date`	Fuse Mediation Router 1.5: The time when the consumer fired

Sample

To set up a route that generates an event every 60 seconds:

```
  from("timer://foo?fixedRate=true&period=60000").to("bean:myBean?method=someMethodName");
```

The above route will generate an event and then invoke the `someMethodName` method on the bean called `myBean` in the Registry such as JNDI or Spring.

And the route in Spring DSL:

```
  <route>
    <from uri="timer://foo?fixedRate=true&eriod=60000"/>
```

```
    <to uri="bean:myBean?method=someMethodName"/>
  </route>
```

See also:

- Quartz on page 389

Chapter 89. Validation

Validation Component

The Validation component performs XML validation of the message body using the JAXP Validation API and based on any of the supported XML schema languages, which defaults to XML Schema[1]

Note that the Jing on page 291 component also supports the following useful schema languages:

- RelaxNG Compact Syntax[2]
- RelaxNG XML Syntax[3]

The MSV on page 361 component also supports RelaxNG XML Syntax[4].

URI format

```
validator:someLocalOrRemoteResource
```

Where **someLocalOrRemoteResource** is some URL to a local resource on the classpath or a full URL to a remote resource or resource on the file system which contains the XSD to validate against. For example:

- `msv:org/foo/bar.xsd`
- `msv:file:../foo/bar.xsd`
- `msv:http://acme.com/cheese.xsd`
- `validator:com/mypackage/myschema.xsd`

Options

Option	Default	Description
`useDom`	`false`	**Fuse Mediation Router 2.0:** Whether `DOMSource`/{{DOMResult}} or `SaxSource`/{{SaxResult}} should be used by the validator.

[1] http://www.w3.org/XML/Schema
[2] http://relaxng.org/compact-tutorial-20030326.html
[3] http://relaxng.org/
[4] http://relaxng.org/

`useSharedSchema true`	**Camel 2.3:** Whether the `Schema` instance should be shared or not. This option is introduced to work around a JDK 1.6.x bug[5]. Xerces should not have this issue.

Example

The following example[6] shows how to configure a route from endpoint **direct:start** which then goes to one of two endpoints, either **mock:valid** or **mock:invalid** based on whether or not the XML matches the given schema (which is supplied on the classpath).

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <doTry>
            <to uri="validator:org/apache/camel/component/validator/schema.xsd"/>
            <to uri="mock:valid"/>
            <doCatch>
                <exception>org.apache.camel.ValidationException</exception>
                <to uri="mock:invalid"/>
            </doCatch>
            <doFinally>
                <to uri="mock:finally"/>
            </doFinally>
        </doTry>
    </route>
</camelContext>
```

[5] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6773084

[6] http://svn.apache.org/repos/asf/camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/component/validator/camelContext.xml

Chapter 90. Velocity

Velocity

The **velocity:** component allows you to process a message using an Apache Velocity[1] template. This can be ideal when using Templating to generate responses for requests.

URI format

```
velocity:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (for example, `file://folder/myfile.vm`).

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

Options

Option	Default	Description
`loaderCache`	`true`	Velocity based file loader cache.
`contentCache`		New option in Fuse Mediation Router 1.4: Cache for the resource content when it is loaded. By default, it's `false` in Fuse Mediation Router 1.x. By default, it's `true` in Fuse Mediation Router 2.x.
`encoding`	`null`	New option in Fuse Mediation Router 1.6: Character encoding of the resource content.
`propertiesFile`	`null`	New option in Camel 2.1: The URI of the properties file which is used for VelocityEngine initialization.

Message Headers

The velocity component sets some headers on the message (you cannot set these yourself):

Header	Description
`org.apache.camel.velocity.resource`	Fuse Mediation Router 1.x: The resource as an `org.springframework.core.io.Resource` object.
`org.apache.camel.velocity.resourceUri`	Fuse Mediation Router 1.x: The **templateName** as a `String` object.

[1] http://velocity.apache.org/

`CamelVelocityResource`	Fuse Mediation Router 2.0: The resource as an `org.springframework.core.io.Resource` object.
`CamelVelocityResourceUri`	Fuse Mediation Router 2.0: The **templateName** as a `String` object.

In Fuse Mediation Router 1.4 headers set during the Velocity evaluation are returned to the message and added as headers. This makes it possible to return values from Velocity to the Message.

For example, to set the header value of `fruit` in the Velocity template `.tm`:

```
$in.setHeader('fruit', 'Apple')
```

The `fruit` header is now accessible from the `message.out.headers`.

Velocity Context

Fuse Mediation Router will provide exchange information in the Velocity context (just a `Map`). The `Exchange` is transfered as:

key	value
`exchange`	The `Exchange` itself.
`headers`	The headers of the In message.
`camelContext`	The Camel Context intance.
`request`	The In message.
`in`	The In message.
`body`	The In message body.
`out`	The Out message (only for InOut message exchange pattern).
`response`	The Out message (only for InOut message exchange pattern).

Hot reloading

The Velocity template resource is, by default, hot reloadable for both file and classpath resources (expanded jar). If you set `contentCache=true`, Fuse Mediation Router will only load the resource once, and thus hot reloading is not possible. This scenario can be used in production, when the resource never changes.

Dynamic templates

Available as of Camel 2.1 Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

Header	Type	Description
CamelVelocityResourceUri	String	**Camel 2.1:** A URI for the template resource to use instead of the endpoint configured.
CamelVelocityTemplate	String	**Camel 2.1:** The template to use instead of the endpoint configured.

Samples

For example you could use something like

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

To use a Velocity template to formulate a response to a message for InOut message exchanges (where there is a `JMSReplyTo` header).

If you want to use InOnly and consume the message and send it to another destination, you could use the following route:

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

And to use the content cache, e.g. for use in production, where the `.vm` template never changes:

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm?contentCache=true").
  to("activemq:Another.Queue");
```

And a file based resource:

```
from("activemq:My.Queue").
  to("velocity:file://myfolder/MyResponse.vm?contentCache=true").
  to("activemq:Another.Queue");
```

In **Camel 2.1** it's possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").
  setHeader("CamelVelocityResourceUri").constant("path/to/my/template.vm").
  to("velocity:dummy");
```

In **Camel 2.1** it's possible to specify a template directly as a header the component should use dynamically via a header, so for example:

```
from("direct:in").
  setHeader("CamelVelocityTemplate").constant("Hi this is a velocity template that can do
templating ${body}").
  to("velocity:dummy");
```

The Email Sample

In this sample we want to use Velocity templating for an order confirmation email. The email template is laid out in Velocity as:

```
Dear ${headers.lastName}, ${headers.firstName}

Thanks for the order of ${headers.item}.

Regards Camel Riders Bookstore
${body}
```

And the java code:

```
private Exchange createLetter() {
    Exchange exchange = context.getEndpoint("direct:a").createExchange();
    Message msg = exchange.getIn();
    msg.setHeader("firstName", "Claus");
    msg.setHeader("lastName", "Ibsen");
    msg.setHeader("item", "Camel in Action");
    msg.setBody("PS: Next beer is on me, James");
    return exchange;
}

@Test
public void testVelocityLetter() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedBodiesReceived("Dear Ibsen, Claus\n\nThanks for the order of Camel in Ac
tion.\n\nRegards Camel Riders Bookstore\nPS: Next beer is on me, James");

    template.send("direct:a", createLetter());

    mock.assertIsSatisfied();
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:a").to("velocity:org/apache/camel/component/velocity/let
```

```
ter.vm").to("mock:result");
        }
    };
}
```

Chapter 91. VM

VM Component

The **vm:** component provides asynchronous SEDA[1] behavior so that messages are exchanged on a BlockingQueue[2] and consumers are invoked in a separate thread pool to the producer.

This component differs from the Seda component in that VM supports communication across CamelContext instances, so you can use this mechanism to communicate across web applications, provided that the `camel-core.jar` is on the `system/boot` classpath.

This component is an extension to the Seda component.

URI format

```
vm:someName[?options]
```

Where **someName** can be any string to uniquely identify the endpoint within the JVM (or at least within the classloader which loaded the camel-core.jar)

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

Options

See the Seda component for options and other important usage as the same rules applies for this Vm component.

Samples

In the route below we send the exchange to the VM queue that is working across CamelContext instances:

```
from("direct:in").bean(MyOrderBean.class).to("vm:order.email");
```

And then in another Camel context such as deployed as in another `.war` application:

```
from("vm:order.email").bean(MyOrderEmailSender.class);
```

See also:

- Seda

[1] http://www.eecs.harvard.edu/~mdw/proj/seda/
[2] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/BlockingQueue.html

Chapter 92. XMPP

XMPP Component

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

URI format

```
xmpp://[login@]hostname[:port][/participant][?Options]
```

The component supports both room based and private person-person conversations. The component supports both producer and consumer (you can get messages from XMPP or send messages to XMPP). Consumer mode supports rooms starting from camel-1.5.0.

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

Options

Name	Description
`room`	If this option is specified, the component will connect to MUC (Multi User Chat). Usually, the domain name for MUC is different from the login domain. For example, if you are `superman@jabber.org` and want to join the `krypton` room, then the room URL is `krypton@conference.jabber.org`. Note the `conference` part.
`user`	User name (without server name). If not specified, anonymous login will be attempted.
`password`	Password.
`resource`	XMPP resource. The default is `Camel`.
`createAccount`	If `true`, an attempt to create an account will be made. Default is `false`.
`participant`	JID (Jabber ID) of person to receive messages. `room` parameter has precedence over `participant`.
`nickname`	Use nickname when joining room. If room is specified and nickname is not, `user` will be used for the nickname.
`serviceName`	**Fuse Mediation Router 1.6/2.0** The name of the service you are connecting to. For Google Talk, this would be `gmail.com`.

Headers and setting Subject or Language

Fuse Mediation Router sets the message IN headers as properties on the XMPP message. You can configure a `HeaderFilterStategy` if you need custom filtering of headers. In **Fuse Mediation Router 1.6.2/2.0** the **Subject** and **Language** of the XMPP message are also set if they are provided as IN headers.

Examples

User `superman` to join room `krypton` at `jabber` server with password, `secret`:

```
xmpp://superman@jabber.org/?room=krypton@conference.jabber.org&password=secret
```

User `superman` to send messages to `joker`:

```
xmpp://superman@jabber.org/joker@jabber.org?password=secret
```

Routing example in Java:

```
from("timer://kickoff?period=10000").
setBody(constant("I will win!\n Your Superman.")).
to("xmpp://superman@jabber.org/joker@jabber.org?password=secret");
```

Consumer configuration, which writes all messages from `joker` into the queue, `evil.talk`.

```
from("xmpp://superman@jabber.org/joker@jabber.org?password=secret").
to("activemq:evil.talk");
```

Consumer configuration, which listens to room messages (supported from camel-1.5.0):

```
from("xmpp://superman@jabber.org/?password=secret&room=krypton@conference.jabber.org").
to("activemq:krypton.talk");
```

Room in short notation (no domain part; for camel-1.5.0+):

```
from("xmpp://superman@jabber.org/?password=secret&room=krypton").
to("activemq:krypton.talk");
```

When connecting to the Google Chat service, you'll need to specify the `serviceName` as well as your credentials (as of **Fuse Mediation Router 1.6/2.0**):

```
// send a message from fromuser@gmail.com to touser@gmail.com
from("direct:start").
        to("xmpp://talk.google.com:5222/touser@gmail.com?serviceName=gmail.com&user=fro
muser&password=secret").
        to("mock:result");
```

Chapter 93. XQuery Endpoint

XQuery

The **xquery:** component allows you to process a message using an XQuery template. This can be ideal when using Templating to generate respopnses for requests.

URI format

```
xquery:templateName
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template.

For example you could use something like this:

```
from("activemq:My.Queue").
  to("xquery:com/acme/mytransform.xquery");
```

To use an XQuery template to formulate a response to a message for InOut message exchanges (where there is a `JMSReplyTo` header).

If you want to use InOnly, consume the message, and send it to another destination, you could use the following route:

```
from("activemq:My.Queue").
  to("xquery:com/acme/mytransform.xquery").
  to("activemq:Another.Queue");
```

Chapter 94. XSLT

XSLT

The **xslt:** component allows you to process a message using an XSLT[1] template. This can be ideal when using Templating to generate respopnses for requests.

URI format

```
xslt:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template. Refer to the Spring Documentation for more detail of the URI syntax[2]

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

Here are some example URIs

URI	Description
`xslt:com/acme/mytransform.xs`	Refers to the file, `com/acme/mytransform.xsl`, on the classpath.
`xslt:file:///foo/bar.xs`	Refers to the file, `/foo/bar.xsl`.
`xslt:http://acme.com/cheese/foo.xsl`	Refers to the remote HTTP resource.

Options

Name	Default Value	Description
`converter`	`null`	Option to override default XmlConverter[3]. Will lookup for the converter in the Registry. The provided converted must be of type org.apache.camel.converter.jaxp.XmlConverter.
`transformerFactory`	`null`	**New added in Fuse Mediation Router 1.6** Option to override default TransformerFactory[4]. Will lookup for the transformerFactory in the Registry. The provided transformer factory must be of type `javax.xml.transform.TransformerFactory`.

[1] http://www.w3.org/TR/xslt
[2] http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/core/io/DefaultResourceLoader.html
[3] http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/converter/jaxp/XmlConverter.html
[4] http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/transform/TransformerFactory.html

`transformerFactoryClass`	`null`	**New added in Fuse Mediation Router 1.6** Option to override default TransformerFactory[5]. Will create a `TransformerFactoryClass` instance and set it to the converter.
`uriResolver`	`null`	**Camel 2.3**: Allows you to use a custom `javax.xml.transformation.URIResolver`. Camel will by default use its own implementation `org.apache.camel.builder.xml.XsltUriResolver` which is capable of loading from classpath.
`resultHandlerFactory`	`null`	**Camel 2.3:** Allows you to use a custom `org.apache.camel.builder.xml.ResultHandlerFactory` which is capable of using custom `org.apache.camel.builder.xml.ResultHandler` types.
`failOnNullBody`	`true`	**Camel 2.3:** Whether or not to throw an exception if the input body is null.
`deleteOutputFile`	`false`	**Camel 2.6:** If you have `output=file` then this option dictates whether or not the output file should be deleted when the Exchange is done processing. For example suppose the output file is a temporary file, then it can be a good idea to delete it after use.
`output`	`string`	**Camel 2.3:** Option to specify which output type to use. Possible values are: `string, bytes, DOM, file`. The first three options are all in memory based, where as `file` is streamed directly to a `java.io.File`. For `file` you **must** specify the filename in the IN header with the key `Exchange.XSLT_FILE_NAME` which is also `CamelXsltFileName`. Also any paths leading to the filename must be created beforehand, otherwise an exception is thrown at runtime.
`contentCache`	`true`	**Camel 2.6:** Cache for the resource content (the stylesheet file) when it is loaded. If set to `false` Camel will reloader the stylesheet file on each message processing. This is good for development.

Using XSLT endpoints

For example you could use something like

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl");
```

To use an XSLT template to forumulate a response for a message for InOut message exchanges (where there is a `JMSReplyTo` header).

[5] http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/transform/TransformerFactory.html

If you want to use InOnly and consume the message and send it to another destination you could use the following route:

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl").
  to("activemq:Another.Queue");
```

Getting Parameters into the XSLT to work with

By default, all headers are added as parameters which are available in the XSLT. To do this you will need to declare the parameter so it is then *useable*.

```
<setHeader headerName="myParam"><constant>42</constant></setHeader>
<to uri="xslt:MyTransform.xsl"/>
```

And the XSLT just needs to declare it at the top level for it to be available:

```
<xsl: ..... >

   <xsl:param name="myParam"/>

    <xsl:template ...>
```

Spring XML versions

To use the above examples in Spring XML you would use something like

```
  <camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
    <route>
      <from uri="activemq:My.Queue"/>
      <to uri="xslt:org/apache/camel/spring/processor/example.xsl"/>
      <to uri="activemq:Another.Queue"/>
    </route>
  </camelContext>
```

There is a test case[6] along with its Spring XML[7] if you want a concrete example.

Using xsl:include

Camel 1.6.2/2.2 or older If you use xsl:include in your XSL files then in Camel 2.2 or older it uses the default `javax.xml.transform.URIResolver` which means it can only lookup files from file system, and its does that relative from the JVM starting folder.

[6] http://svn.apache.org/repos/asf/camel/trunk/components/camel-spring/src/test/java/org/apache/camel/spring/processor/XsltTest.java
[7] http://svn.apache.org/repos/asf/camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/XsltTest-context.xml

For example this include:

```
<xsl:include href="staff_template.xsl"/>
```

Will lookup the `staff_tempkalte.xsl` file from the starting folder where the application was started.

Camel 1.6.3/2.3 or newer Now Camel provides its own implementation of `URIResolver` which allows Camel to load included files from the classpath and more intelligent than before.

For example this include:

```
<xsl:include href="staff_template.xsl"/>
```

Will now be located relative from the starting endpoint, which for example could be:

```
.to("xslt:org/apache/camel/component/xslt/staff_include_relative.xsl")
```

Which means Camel will locate the file in the **classpath** as `org/apache/camel/component/xslt/staff_template.xsl`. This allows you to use xsl include and have xsl files located in the same folder such as we do in the example `org/apache/camel/component/xslt`.

You can use the following two prefixes `classpath:` or `file:` to instruct Camel to look either in classpath or file system. If you omit the prefix then Camel uses the prefix from the endpoint configuration. If that neither has one, then classpath is assumed.

You can also refer back in the paths such as

```
    <xsl:include href="../staff_other_template.xsl"/>
```

Which then will resolve the xsl file under `org/apache/camel/component`.

Index