Table of Contents

CHAPTER 1

°°°°

Introduction

Apache Camel is a powerful Spring based Integration Framework.
Camel implements the Enterprise Integration Patterns allowing you to configure routing and mediation rules in either a Java based Domain Specific Language (or Fluent API) or via Spring based Xml Configuration files. Either approaches mean you get smart completion of routing rules in your IDE whether in your Java or XML editor.

Apache Camel uses URIs so that it can easily work directly with any kind of Transport or messaging model such as HTTP, ActiveMQ, JMS, JBI, SCA, MINA or CXF Bus API together with working with pluggable Data Format options. Apache Camel is a small library which has minimal dependencies for easy embedding in any Java application.

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

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

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

CHAPTER 2

°°°°

Getting Started with Apache Camel

THE *ENTERPRISE INTEGRATION PATTERNS* (EIP) BOOK

The purpose of a "patterns" book is not to advocate new techniques that the authors have invented, but rather to document existing best practices within a particular field. By doing this, the authors of a patterns book hope to spread knowledge of best practices and promote a vocabulary for discussing architectural designs.
One of the most famous patterns books is *Design Patterns: Elements of Reusable Object-oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Some people refer to this as the "gang of four" book, partly to distinguish this book from other books that use "Design Patterns" in their titles and, perhaps, partly because they cannot remember the names of all four authors.
Since the publication of *Design Patterns*, many other patterns books, of varying quality, have been written. One famous patterns book is called *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* by Gregor Hohpe and Bobby Woolfe. It is common for people to refer to this book as *EIP*, which is an acronym of its title. As the subtitle of EIP suggests, the book focusses on design patterns for asynchronous messaging systems. The book discusses 65 patterns. Each pattern is given a textual name and most are also given a graphical symbol. The graphical symbols are intended to be used in architectural diagrams.

THE CAMEL PROJECT

Camel (http://activemq.apache.org/camel/) is an open-source, Java-based project that is a part of the Apache *ActiveMQ* project. Camel provides a class library that, according to its documentation, can be used to implement 31 design patterns in the EIP book. I am not sure why the Camel documentation discusses only 31 of the 65 EIP design patterns. Perhaps this is due to incomplete documentation. Or perhaps it means that the Camel project, which is less than 1 year old at the time of writing, is not yet as feature rich as the EIP book.
Because Camel implements many of the design patterns in the EIP book, it would be a good idea for people who work with Camel to read the EIP book.

ONLINE DOCUMENTATION FOR CAMEL

The Camel project was started in early 2007. At the time of writing, the Camel project is too young for there to be published books available on how to use Camel. Instead, the only source of documentation seems to the documentation page on the Apache Camel website.

Problems with Camel's online documentation

Currently, the online documentation for the Apache Camel project suffers from two problems. First, the documentation is incomplete. Second, there is no clearly specified reading order to the documentation. For example, there is no table of contents. Instead, documentation is fragmented over a collection of 60+ web pages, and hypertext links haphazardly tie these web pages to each other. This documentation might suffice as reference material for people already familiar with Camel but it does not qualify as a tutorial for beginners.
The problems with the documentation are unlikely to be due to, say, its author(s) lacking writing ability. Rather, it is more likely that the problems are due to the author(s) lack of time. I expect Camel's documentation will improve over time. I am writing this overview of Camel to partially counter some of the problems that currently afflict the Camel documentation. In particular, this document aims to serve as a (so far, incomplete) "beginner's guide to Camel". As such, this document tries to complement, rather than compete with, the online Camel documentation.

A useful tip for navigating the online documentation

There is one useful hint I can provide for reading the online Camel documentation. Each documentation page has a logo at the top, and immediately underneath this is a think reddish bar that contains some hypertext links. The Hypertext links on left side of this reddish bar indicate your position in documentation. For example, If you are on the "Languages" documentation page then the left-hand side of the reddish bar contains the following links.

```
Apache Camel > Documentation > Architecture > Languages
```

As you might expect, clicking on "Apache Camel" takes you back to the home page of the Apache Camel project, and clicking on "Documentation" takes you to the main documentation page. You can interpret the "Architeture" and "Languages" buttons as indicating you are in the "Languages" section of the "Architecture" chapter. Doing this gives you at least some sense of where you are within the documentation. If you are patient then you can spend a few hours clicking on all the hypertext links you can find in the documentation pages, bookmark each page with a hierarchical name (for example, you might bookmark the above page with the name "Camel Ð Arch Ð Languages") and then you can use your bookmarks to serve as a primitive table of contents for the online Camel documentation.

ONLINE JAVADOC DOCUMENTATION

The Apache Camel website provides Javadoc documentation. It is important to note that the Javadoc documentation is spread over several *independent* Javadoc hierarchies rather than being all contained in a single Javadoc hierarchy. In particular, there is one Javadoc hierarchy for the *core* APIs of Camel, and a separate Javadoc hierarchy for each communications technology supported by Camel. For example, if you will be using Camel with ActiveMQ and FTP then you need to look at the Javadoc hierarchies for the core API, ActiveMQ API and FTP API.

CONCEPTS AND TERMINOLOGY FUNDAMENTAL TO CAMEL

I said in Section 3.1 ("Problems with Camel's online documentation") that the online Camel documentation does not provide a tutorial for beginners. Because of this, in this section I try to explain some of the concepts and terminology that are fundamental to Camel. This section is not a complete Camel tutorial, but it is a first step in that direction.

Endpoint

The term *endpoint* is often used when talking about inter-process communication. For example, in client-server communication, the client is one endpoint and the server is the other endpoint. Depending on the context, an endpoint might refer to an *address*, such as a host:port pair for TCP-based communication, or it might refer to a *software entity* that is contactable at that address. For example, if somebody uses "www.example.com:80" as an example of an endpoint, they might be referring to the actual port at that host name (that is, an address), or they might be referring to the web server (that is, software contactable at that address). Often, the distinction between the address and software contactable at that address is not an important one.

Some middleware technologies make it possible for several software entities to be contactable at the same physical address. For example, CORBA is an object-oriented, remote-procedure-call (RPC) middleware standard. If a CORBA server process contains several objects then a client can communicate with any of these objects at the same *physical* address (host:port), but a client communicates with a particular object via that object's *logical* address (called an *IOR* in CORBA terminology), which consists of the physical address (host:port) plus an id that uniquely identifies the object within its server process. (An IOR contains some additional information that is not relevant to this present discussion.) When talking about CORBA, some people may use the term "endpoint" to refer to a CORBA server's *physical address*, while other people may use the term to refer to the *logical address* of a single CORBA object, and other people still might use the term to refer to any of the following:

- The physical address (host:port) of the CORBA server process
- The logical address (host:port plus id) of a CORBA object.
- The CORBA server process (a relatively heavyweight software entity)
- A CORBA object (a lightweight software entity)

  Because of this, you can see that the term *endpoint* is ambiguous in at least two ways.

First, it is ambiguous because it might refer to an address or to a software entity contactable at that address. Second, it is ambiguous in the *granularity* of what it refers to: a heavyweight versus lightweight software entity, or physical address versus logical address. It is useful to understand that different people use the term *endpoint* in slightly different (and hence ambiguous) ways because Camel's usage of this term might be different to whatever meaning you had previously associated with the term. Camel provides out-of-the-box support for endpoints implemented with many different communication technologies. Here are some examples of the Camel-supported endpoint technologies.

- A JMS queue.
- A web service.
- A file. A file may sound like an unlikely type of endpoint, until you realize that in some systems one application might write information to a file and, later, another application might read that file.
- An FTP server.
- An email address. A client can send a message to an email address, and a server can read an incoming message from a mail server.
- A POJO (plain old Java object).
  In a Camel-based application, you create (Camel wrappers around) some endpoints and connect these endpoints with *routes*, which I will discuss later in Section 4.8 ("Routes, RouteBuilders and Java DSL"). Camel defines a Java interface called `Endpoint`. Each Camel-supported endpoint has a class that implements this `Endpoint` interface. As I discussed in Section 3.3 ("Online Javadoc documentation"), Camel provides a separate Javadoc hierarchy for each communications technology supported by Camel. Because of this, you will find documentation on, say, the `JmsEndpoint` class in the JMS Javadoc hierarchy, while documentation for, say, the `FtpEndpoint` class is in the FTP Javadoc hierarchy.

CamelContext

A `CamelContext` object represents the Camel runtime system. You typically have one `CamelContext` object in an application. A typical application executes the following steps.

1. Create a `CamelContext` object.
2. Add endpoints Ð and possibly Components, which are discussed in Section 4.5 ("Components") Ð to the `CamelContext` object.
3. Add routes to the `CamelContext` object to connect the endpoints.
4. Invoke the `start()` operation on the `CamelContext` object. This starts Camel-internal threads that are used to process the sending, receiving and processing of messages in the endpoints.
5. Eventually invoke the `stop()` operation on the `CamelContext` object. Doing this gracefully stops all the endpoints and Camel-internal threads.
   Note that the `CamelContext.start()` operation does not block indefinitely.

Rather, it starts threads internal to each `Component` and `Endpoint` and then `start()` returns. Conversely, `CamelContext.stop()` waits for all the threads internal to each `Endpoint` and `Component` to terminate and then `stop()` returns.
If you neglect to call `CamelContext.start()` in your application then messages will not be processed because internal threads will not have been created.
If you neglect to call `CamelContext.stop()` before terminating your application then the application may terminate in an inconsistent state. If you neglect to call `CamelContext.stop()` in a JUnit test then the test may fail due to messages not having had a chance to be fully processed.

CamelTemplate

Camel used to have a class called `CamelClient`, but this was renamed to be `CamelTemplate` to be similar to a naming convention used in some other open-source projects, such as the `TransactionTemplate` and `JmsTemplate` classes in Spring.
The `CamelTemplate` class is a thin wrapper around the `CamelContext` class. It has methods that send a `Message` or `Exchange` Ð both discussed in Section 4.6 ("Message and Exchange")) Ð to an `Endpoint` Ð discussed in Section 4.1 ("Endpoint"). This provides a way to enter messages into source endpoints, so that the messages will move along routes Ð discussed in Section 4.8 ("Routes, RouteBuilders and Java DSL") Ð to destination endpoints.

The Meaning of URL, URI, URN and IRI

Some Camel methods take a parameter that is a *URI* string. Many people know that a URI is "something like a URL" but do not properly understand the relationship between URI and URL, or indeed its relationship with other acronyms such as IRI and URN.
Most people are familiar with *URLs* (uniform resource locators), such as "http://...", "ftp://...", "mailto:...". Put simply, a URL specifies the *location* of a resource.
A *URI* (uniform resource identifier) is a URL *or* a URN. So, to fully understand what URI means, you need to first understand what is a URN.
URN is an acronym for *uniform resource name*. There are may "unique identifier" schemes in the world, for example, ISBNs (globally unique for books), social security numbers (unique within a country), customer numbers (unique within a company's customers database) and telephone numbers. Each "unique identifier" scheme has its own notation. A URN is a wrapper for different "unique identifier" schemes. The syntax of a URN is "urn:<scheme-name>:<unique-identifier>". A URN uniquely identifies a *resource*, such as a book, person or piece of equipment. By itself, a URN does not specify the *location* of the resource. Instead, it is assumed that a *registry* provides a mapping from a resource's URN to its location. The URN specification does not state what form a registry takes, but it might be a database, a server application, a wall chart or anything else that is convenient. Some hypothetical examples of URNs are "urn:employee:08765245", "urn:customer:uk:3458:hul8" and "urn:foo:0000-0000-9E59-0000-5E-2". The <scheme-name> ("employee", "customer" and "foo"

in these examples) part of a URN implicitly defines how to parse and interpret the <unique-identifier> that follows it. An arbitrary URN is meaningless unless: (1) you know the semantics implied by the <scheme-name>, and (2) you have access to the registry appropriate for the <scheme-name>. A registry does not have to be public or globally accessible. For example, "urn:employee:08765245" might be meaningful only within a specific company.
To date, URNs are not (yet) as popular as URLs. For this reason, URI is widely misused as a synonym for URL.
IRI is an acronym for *internationalized resource identifier*. An IRI is simply an internationalized version of a URI. In particular, a URI can contain letters and digits in the US-ASCII character set, while a IRI can contain those same letters and digits, and *also* European accented characters, Greek letters, Chinese ideograms and so on.

Components

Component is confusing terminology; *EndpointFactory* would have been more appropriate because a `Component` is a factory for creating `Endpoint` instances. For example, if a Camel-based application uses several JMS queues then the application will create one instance of the `JmsComponent` class (which implements the `Component` interface), and then the application invokes the `createEndpoint()` operation on this `JmsComponent` object several times. Each invocation of `JmsComponent.createEndpoint()` creates an instance of the `JmsEndpoint` class (which implements the `Endpoint` interface). Actually, application-level code does not invoke `Component.createEndpoint()` directly. Instead, application-level code normally invokes `CamelContext.getEndpoint()`; internally, the `CamelContext` object finds the desired `Component` object (as I will discuss shortly) and then invokes `createEndpoint()` on it.
Consider the following code.

```
myCamelContext.getEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword");
```

The parameter to `getEndpoint()` is a URI. The URI *prefix* (that is, the part before ":") specifies the name of a component. Internally, the `CamelContext` object maintains a mapping from names of components to `Component` objects. For the URI given in the above example, the `CamelContext` object would probably map the `pop3` prefix to an instance of the `MailComponent` class. Then the `CamelContext` object invokes `createEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword'` on that `MailComponent` object. The `createEndpoint()` operation splits the URI into its component parts and uses these parts to create and configure an `Endpoint` object.
In the previous paragraph, I mentioned that a `CamelContext` object maintains a mapping from component names to `Component` objects. This raises the question of how this map is populated with named `Component` objects. There are two ways of populating the map. The first way is for application-level code to invoke `CamelContext.addComponent(String`

componentName, Component component). The example below shows a single MailComponent object being registered in the map under 3 different names.

```
Component mailComponent = new org.apache.camel.component.mail.MailComponent();
myCamelContext.addComponent("pop3", mailComponent);
myCamelContext.addComponent("imap", mailComponent);
myCamelContext.addComponent("smtp", mailComponent);
```

The second (and preferred) way to populate the map of named Component objects in the CamelContext object is to let the CamelContext object perform lazy initialization. This approach relies on developers following a convention when they write a class that implements the Component interface. I illustrate the convention by an example. Let's assume you write a class called com.example.myproject.FooComponent and you want Camel to automatically recognize this by the name "foo". To do this, you have to write a properties file called "META-INF/services/org/apache/camel/component/foo" (without a ".properties" file extension) that has a single entry in it called class, the value of which is the fully-scoped name of your class. This is shown below.

Listing 1. META-INF/services/org/apache/camel/component/foo

```
class=com.example.myproject.FooComponent
```

If you want Camel to also recognize the class by the name "bar" then you write another properties file in the same directory called "bar" that has the same contents. Once you have written the properties file(s), you create a jar file that contains the com.example.myproject.FooComponent class and the properties file(s), and you add this jar file to your CLASSPATH. Then, when application-level code invokes createEndpoint("foo:...") on a CamelContext object, Camel will find the "foo"" properties file on the CLASSPATH, get the value of the class property from that properties file, and use reflection APIs to create an instance of the specified class.
As I said in Section 4.1 ("Endpoint"), Camel provides out-of-the-box support for numerous communication technologies. The out-of-the-box support consists of classes that implement the Component interface plus properties files that enable a CamelContext object to populate its map of named Component objects.
Earlier in this section I gave the following example of calling CamelContext.getEndpoint().

```
myCamelContext.getEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword");
```

When I originally gave that example, I said that the parameter to getEndpoint() was a URI. I said that because the online Camel documentation and the Camel source code both claim the parameter is a URI. In reality, the parameter is restricted to being a URL. This is because when Camel extracts the component name from the parameter, it looks for the first ":", which is a simplistic algorithm. To understand why, recall from Section 4.4 ("The Meaning of URL, URI, URN and IRI") that a URI can be a URL *or* a URN. Now consider the following calls to `getEndpoint`.

```
myCamelContext.getEndpoint("pop3:...");
myCamelContext.getEndpoint("jms:...");
myCamelContext.getEndpoint("urn:foo:...");
myCamelContext.getEndpoint("urn:bar:...");
```

Camel identifies the components in the above example as "pop3", "jms", "urn" and "urn". It would be more useful if the latter components were identified as "urn:foo" and "urn:bar" or, alternatively, as "foo" and "bar" (that is, by skipping over the "urn:" prefix). So, in practice you must identify an endpoint with a URL (a string of the form "<scheme>:...") rather than with a URN (a string of the form "urn:<scheme>:..."). This lack of proper support for URNs means the you should consider the parameter to `getEndpoint()` as being a URL rather than (as claimed) a URI.

Message and Exchange

The `Message` interface provides an abstraction for a single message, such as a request, reply or exception message.
There are concrete classes that implement the `Message` interface for each Camel-supported communications technology. For example, the `JmsMessage` class provides a JMS-specific implementation of the `Message` interface. The public API of the `Message` interface provides get- and set-style methods to access the *message id*, *body* and individual *header* fields of a messge.
The `Exchange` interface provides an abstraction for an exchange of messages, that is, a request message and its corresponding reply or exception message. In Camel terminology, the request, reply and exception messages are called *in*, *out* and *fault* messages.
There are concrete classes that implement the `Exchange` interface for each Camel-supported communications technology. For example, the `JmsExchange` class provides a JMS-specific implementation of the `Exchange` interface. The public API of the `Exchange` interface is quite limited. This is intentional, and it is expected that each class that implements this interface will provide its own technology-specific operations.
Application-level programmers rarely access the `Exchange` interface (or classes that implement it) directly. However, many classes in Camel are generic types that are instantiated on (a class that implements) `Exchange`. Because of this, the `Exchange` interface appears a lot in the generic signatures of classes and methods.

Processor

The `Processor` interface represents a class that processes a message. The signature of this interface is shown below.

Listing 2. Processor

```
package org.apache.camel;
public interface Processor {
    void process(Exchange exchange) throws Exception;
}
```

Notice that the parameter to the `process()` method is an `Exchange` rather than a `Message`. This provides flexibility. For example, an implementation of this method initially might call `exchange.getIn()` to get the input message and process it. If an error occurs during processing then the method can call `exchange.setException()`.
An application-level developer might implement the `Processor` interface with a class that executes some business logic. However, there are many classes in the Camel library that implement the `Processor` interface in a way that provides support for a design pattern in the EIP book. For example, `ChoiceProcessor` implements the message router pattern, that is, it uses a cascading if-then-else statement to route a message from an input queue to one of several output queues. Another example is the `FilterProcessor` class which discards messages that do not satisfy a stated *predicate* (that is, condition).

Routes, RouteBuilders and Java DSL

A *route* is the step-by-step movement of a `Message` from an input queue, through arbitrary types of decision making (such as filters and routers) to a destination queue (if any). Camel provides two ways for an application developer to specify routes. One way is to specify route information in an XML file. A discussion of that approach is outside the scope of this document. The other way is through what Camel calls a Java *DSL* (domain-specific language).

Introduction to Java DSL

For many people, the term "domain-specific language" implies a compiler or interpreter that can process an input file containing keywords and syntax specific to a particular domain. This is *not* the approach taken by Camel. Camel documentation consistently uses the term "Java DSL" instead of "DSL", but this does not entirely avoid potential confusion. The Camel "Java DSL" is a class library that can be used in a way that looks almost like a DSL, except that it has a bit of Java syntactic baggage. You can see this in the example below. Comments afterwards explain some of the constructs used in the example.

Listing 3. Example of Camel's "Java DSL"

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

The first line in the above example creates an object which is an instance of an anonymous subclass of `RouteBuilder` with the specified `configure()` method.
The `CamelContext.addRoutes(RouterBuilder builder)` method invokes `builder.setContext(this)` Ð so the `RouteBuilder` object knows which `CamelContext` object it is associated with Ð and then invokes `builder.configure()`. The body of `configure()` invokes methods such as `from()`, `filter()`, `choice()`, `when()`, `isEqualTo()`, `otherwise()` and `to()`.
The `RouteBuilder.from(String uri)` method invokes `getEndpoint(uri)` on the `CamelContext` associated with the `RouteBuilder` object to get the specified `Endpoint` and then puts a `FromBuilder` "wrapper" around this `Endpoint`. The `FromBuilder.filter(Predicate predicate)` method creates a `FilterProcessor` object for the `Predicate` (that is, condition) object built from the `header("foo").isEqualTo("bar")` expression. In this way, these operations incrementally build up a `Route` object (with a `RouteBuilder` wrapper around it) and add it to the `CamelContext` object associated with the `RouteBuilder`.

Critique of Java DSL

The online Camel documentation compares Java DSL favourably against the alternative of configuring routes and endpoints in a XML-based Spring configuration file. In particular, Java DSL is less verbose than its XML counterpart. In addition, many integrated development environments (IDEs) provide an auto-completion feature in their editors. This auto-completion feature works with Java DSL, thereby making it easier for developers to write Java DSL.
However, there is another option that the Camel documentation neglects to consider: that of writing a parser that can process DSL stored in, say, an external file. Currently, Camel does not provide such a DSL parser, and I do not know if it is on the "to do" list of the Camel maintainers. I think that a DSL parser would offer a significant benefit over the current Java DSL. In particular, the DSL would have a syntactic definition that could be expressed in a relatively short BNF form. The effort required by a Camel user to learn how to use DSL by reading this BNF would almost certainly be significantly less than the effort currently required to study the API of the `RouterBuilder` classes.

CHAPTER 3

° ° ° °

Architecture

Camel uses a Java based Routing Domain Specific Language (DSL) or an Xml Configuration to configure routing and mediation rules which are added to a CamelContext to implement the various Enterprise Integration Patterns.
At a high level Camel consists of a CamelContext which contains a collection of Component instances. A Component is essentially a factory of Endpoint instances. You can explicitly configure Component instances in Java code or an IoC container like Spring or Guice, or they can be auto-discovered using URIs.

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

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

- Bean Language
- the unified EL from JSP and JSF
- JXPath
- OGNL
- Scripting Languages such as
    - BeanShell
    - JavaScript
    - Groovy
    - Python
    - PHP
    - Ruby
- Simple
- SQL
- XPath
- XQuery

URIS

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

Current Supported URIs

Component / ArtifactId / URI	Description
ActiveMQ / activemq-camel `activemq:[topic:]destinationName`	For JMS Messaging with Apache ActiveMQ
ActiveMQ Journal / activemq-core `activemq.journal:directory-on-filesystem`	Uses ActiveMQ's fast disk journaling implementation to store message bodies in a rolling log file
AMQP / camel-amqp `amqp:[topic:]destinationName`	For Messaging with AMQP protocol
Atom / camel-atom `atom:uri`	Working with Apache Abdera for atom integration, such as consuming an atom feed.
Bean / camel-core `bean:beanName[?methodName=someMethod]`	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).
CXF / camel-cxf `cxf:address[?serviceClass=...]`	Working with Apache CXF for web services integration
DataSet / camel-core `dataset:name`	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 / camel-core `direct:name`	Direct invocation of the consumer from the producer so that single threaded (non-SEDA) in VM invocation is performed

Esper / camel-esper in camel-extra `esper:name`	Working with the Esper Library for Event Stream Processing
Event / camel-spring `event://default`	Working with Spring ApplicationEvents
File / camel-core `file://nameOfFileOrDirectory`	Sending messages to a file or polling a file or directory
FIX / camel-fix in FUSE `fix://configurationResource`	Sends or receives messages using the FIX protocol
Flatpack / camel-flatpack `flatpack:[fixed|delim]:configFile`	Processing fixed width or delimited files or messages using the FlatPack library
FTP / camel-ftp `ftp://host[:port]/fileName`	Sending and receiving files over FTP
Hibernate / camel-hibernate in camel-extra `hibernate://entityName`	For using a database as a queue via the Hibernate library
HTTP / camel-http `http://hostname[:port]`	For calling out to external HTTP servers
iBATIS / camel-ibatis `ibatis://sqlOperationName`	Performs a query, poll, insert, update or delete in a relational database using Apache iBATIS
IMap / camel-mail `imap://hostname[:port]`	Receiving email using IMap
IRC / camel-irc `irc:host[:port]/#room`	For IRC communication

JavaSpace / camel-javaspace in FUSE `javaspace:jini://host?spaceName=mySpace?...`	Sending and receiving messages through JavaSpace
JBI / servicemix-camel `jbi:serviceName`	For JBI integration such as working with Apache ServiceMix
JCR / camel-jcr `jcr://user:password@repository/path/to/node`	Storing a message in a JCR (JSR-170) compliant repository like Apache Jackrabbit
JDBC / camel-jdbc `jdbc:dataSourceName?options`	For performing JDBC queries and operations
Jetty / camel-jetty `jetty:url`	For exposing services over HTTP
JMS / camel-jms `jms:[topic:]destinationName`	Working with JMS providers
JPA / camel-jpa `jpa://entityName`	For using a database as a queue via the JPA specification for working with OpenJPA, Hibernate or TopLink
List / camel-core `list:someName`	Provdes 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.
Log / camel-core `log:loggingCategory[?level=ERROR]`	Uses Jakarta Commons Logging to log the message exchange to some underlying logging system like log4j
Mail / camel-mail `mail://user-info@host:port`	Sending and receiving email

MINA / camel-mina `[tcp|udp|multicast]:host[:port]`	Working with Apache MINA
Mock / camel-core `mock:name`	For testing routes and mediation rules using mocks
MSMQ / camel-msmq in FUSE `msmq:msmqQueueName`	Sending and receiving messages with Microsoft Message Queuing
MSV / camel-msv `msv:someLocalOrRemoteResource`	Validates the payload of a message using the MSV Library
Multicast / camel-mina `multicast://host:port`	Working with TCP protocols using Apache MINA
Pojo / camel-core `pojo:name`	**Deprecated**. It is now an alias to the Bean component.
POP / camel-mail `pop3://user-info@host:port`	Receiving email using POP3 and JavaMail
Quartz / camel-quartz `quartz://groupName/timerName`	Provides a scheduled delivery of messages using the Quartz scheduler
Queue / camel-core `queue:name`	**Deprecated**.Ê It is now an alias toÊthe SEDA component.
Ref / camel-core `ref:name`	Component for lookup of existing endpoints bound in the Registry.
RMI / camel-rmi `rmi://host[:port]`	Working with RMI

RNC / camel-jing `rnc:/relativeOrAbsoluteUri`	Validates the payload of a message using RelaxNG Compact Syntax
RNG / camel-jing `rng:/relativeOrAbsoluteUri`	Validates the payload of a message using RelaxNG
SEDA / camel-core `seda:name`	Used to deliver messages to a java.util.concurrent.BlockingQueue, useful when creating SEDA style processing pipelines within the same CamelContext
SFTP / camel-ftp `sftp://host[:port]/fileName`	Sending and receiving files over SFTP
SMTP / camel-mail `smtp://user-info@host[:port]`	Sending email using SMTP and JavaMail
SpringIntegration / camel-spring-integration `spring-integration:defaultChannelName`	The bridge component of Camel and Spring Integration
SQL / camel-sql `sql:select * from table where id=#`	Performing SQL queries using JDBC
Stream / camel-stream `stream:[in|out|err|file]`	Read or write to an input/output/error/file stream rather like unix pipes
StringTemplate / camel-stringtemplate `string-template:someTemplateResource`	Generates a response using a String Template
TCP / camel-mina `tcp://host:port`	Working with TCP protocols using Apache MINA

Test / camel-spring `test:expectedMessagesEndpointUri`	Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint
Timer / camel-core `timer://name`	A timer endpoint
UDP / camel-mina `udp://host:port`	Working with UDP protocols using Apache MINA
Validation / camel-spring `validation:someLocalOrRemoteResource`	Validates the payload of a message using XML Schema and JAXP Validation
Velocity / camel-velocity `velocity:someTemplateResource`	Generates a response using an Apache Velocity template
VM / camel-core `vm:name`	Used to deliver messages to a java.util.concurrent.BlockingQueue, useful when creating SEDA style processing pipelines within the same JVM
XMPP / camel-xmpp `xmpp://host:port/room`	Working with XMPP and Jabber
XQuery / camel-saxon `xquery:someXQueryResource`	Generates a response using an XQuery template
XSLT / camel-spring `xslt:someTemplateResource`	Generates a response using an XSLT template

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

CHAPTER 4

°°°°

Enterprise Integration Patterns

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

PATTERN INDEX

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

Messaging Systems

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

Messaging Channels

Point to Point Channel	How can the caller be sure that exactly one receiver will receive the document or perform the call?
Publish Subscribe Channel	How can the sender broadcast an event to all interested receivers?
Dead Letter Channel	What will the messaging system do with a message it cannot deliver?
Guaranteed Delivery	How can the sender make sure that a message will be delivered, even if the messaging system fails?
Message Bus	What is an architecture that enables separate applications to work together, but in a de-coupled fashion such that applications can be easily added or removed without affecting the others?

Message Construction

Correlation Identifier	How does a requestor that has received a reply know which request this is the reply for?

Message Routing

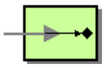
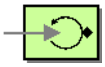
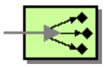
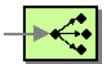
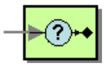
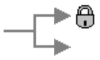
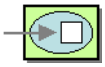
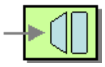
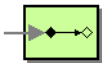
Content Based Router	How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?
Message Filter	How can a component avoid receiving uninteresting messages?
Recipient List	How do we route a message to a list of dynamically specified recipients?
Splitter	How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?
Aggregator	How do we combine the results of individual, but related messages so that they can be processed as a whole?

	Resequencer	How can we get a stream of related but out-of-sequence messages back into the correct order?
	Routing Slip	How do we route a message consecutively through a series of processing steps when the sequence of steps is not known at design-time and may vary for each message?
Unable to render embedded object: File (clear.png) not found.	Throttler	How can I throttle messages to ensure that a specific endpoint does not get overloaded, or we don't exceed an agreed SLA with some external service?
Unable to render embedded object: File (clear.png) not found.	Delayer	How can I delay the sending of a message?
Unable to render embedded object: File (clear.png) not found.	Load Balancer	How can I balance load across a number of endpoints?
Unable to render embedded object: File (clear.png) not found.	Multicast	How can I route a message to a number of endpoints at the same time?

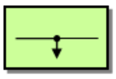
Message Transformation

	Content Enricher	How do we communicate with another system if the message originator does not have all the required data items available?
	Content Filter	How do you simplify dealing with a large message, when you are interested only in a few data items?
	Normalizer	How do you process messages that are semantically equivalent, but arrive in a different format?

Messaging Endpoints

Unable to render embedded object: File (clear.png) not found.	Messaging Mapper	How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?
	Event Driven Consumer	How can an application automatically consume messages as they become available?
	Polling Consumer	How can an application consume a message when the application is ready?
	Competing Consumers	How can a messaging client process multiple messages concurrently?
	Message Dispatcher	How can multiple consumers on a single channel coordinate their message processing?
	Selective Consumer	How can a message consumer select which messages it wishes to receive?
	Durable Subscriber	How can a subscriber avoid missing messages while it's not listening for them?
Unable to render embedded object: File (clear.png) not found.	Idempotent Consumer	How can a message receiver deal with duplicate messages?
	Transactional Client	How can a client control its transactions with the messaging system?
	Messaging Gateway	How do you encapsulate access to the messaging system from the rest of the application?
	Service Activator	How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?

System Management

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

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

CHAPTER 5

° ° ° °

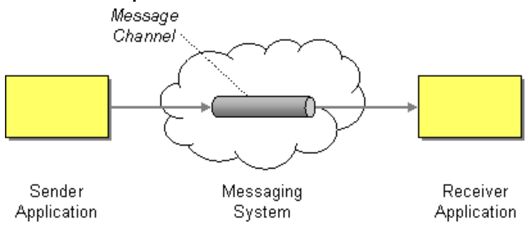
Pattern Appendix

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

MESSAGING SYSTEMS

Message Channel

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



For more details see

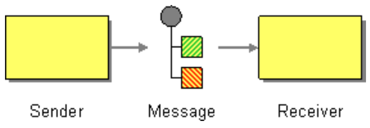
- Message
- Message Endpoint

Using This Pattern

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

Message

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



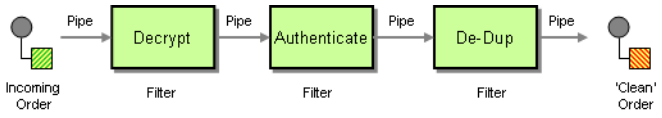
To support various message exchange patterns like one way event messages and request-response messages Camel uses an Exchange interface which is used to handle either oneway messages with a single inbound Message, or request-reply where there is an inbound and outbound message.

Using This Pattern

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

Pipes and Filters

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



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

Using Routing Logic

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

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

Though pipeline is the default mode of operation when you specify multiple outputs in Camel. The opposite to pipeline is multicast; which fires the same message into each of its outputs. (See the example below).

In Spring XML you can use the <pipeline/> element as of 1.4.0 onwards

```
<route>
  <from uri="activemq:SomeQueue"/>
  <pipeline>
    <bean ref="foo"/>
    <bean ref="bar"/>
    <to uri="activemq:OutputQueue"/>
  </pipeline>
</route>
```

In the above the pipeline element is actually unnecessary, you could use this...

```
<route>
  <from uri="activemq:SomeQueue"/>
  <bean ref="foo"/>
  <bean ref="bar"/>
  <to uri="activemq:OutputQueue"/>
</route>
```

Its just a bit more explicit. However if you wish to use <multicast/> to avoid a pipeline - to send the same message into multiple pipelines - then the <pipeline/> element comes into its own.

```
<route>
  <from uri="activemq:SomeQueue"/>
  <multicast>
    <pipeline>
      <bean ref="something"/>
      <to uri="log:Something"/>
    </pipeline>
    <pipeline>
      <bean ref="foo"/>
      <bean ref="bar"/>
      <to uri="activemq:OutputQueue"/>
    </pipeline>
  </multicast>
</route>
```

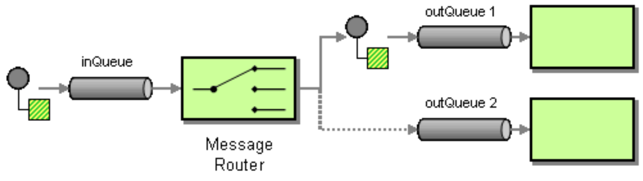
In the above example we are routing from a single Endpoint to a list of different endpoints specified using URIs. If you find the above a bit confusing, try reading about the Architecture or try the Examples

Using This Pattern

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

Message Router

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



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

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").choice().when(header("foo").isEqualTo("bar")).to("seda:b")

.when(header("foo").isEqualTo("cheese")).to("seda:c").otherwise().to("seda:d");
    }
};
```

Using the Spring XML Extensions

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <choice>
      <when>
        <xpath>$foo = 'bar'</xpath>
        <to uri="seda:b"/>
      </when>
      <when>
        <xpath>$foo = 'cheese'</xpath>
        <to uri="seda:c"/>
      </when>
      <otherwise>
        <to uri="seda:d"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

Choice without otherwise

If you use a `choice` without adding an `otherwise`, any unmatched exchanges will be dropped by default. If you prefer to have an exception for an unmatched exchange, you can add a `throwFault` to the `otherwise`.

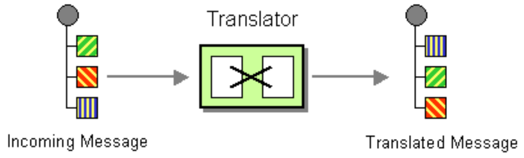
```
....otherwise().throwFault("No matching when clause found on choice block");
```

Using This Pattern

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

Message Translator

Camel supports the Message Translator from the EIP patterns by using an arbitrary Processor in the routing logic, by using a bean to perform the transformation, or by using transform() in the DSL. You can also use a Data Format to marshal and unmarshal messages in different encodings.



Using the Fluent Builders

You can transform a message using Camel's Bean Integration to call any method on a bean in your Registry such as your Spring XML configuration file as follows

```
from("activemq:SomeQueue").
  beanRef("myTransformerBean", "myMethodName").
  to("mqseries:AnotherQueue");
```

Where the "myTransformerBean" would be defined in a Spring XML file or defined in JNDI etc. You can omit the method name parameter from beanRef() and the Bean Integration will try to deduce the method to invoke from the message exchange.

or you can add your own explicit Processor to do the transformation

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

or you can use the DSL to explicitly configure the transformation

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

Use Spring XML

You can also use Spring XML Extensions to do a transformation. Basically any Expression language can be substituted inside the transform element as shown below

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <transform>
      <simple>${in.body} extra data!</simple>
    </transform>
    <to uri="mock:end"/>
  </route>
</camelContext>
```

Or you can use the Bean Integration to invoke a bean

```
<route>
  <from uri="activemq:Input"/>
  <bean ref="myBeanName" method="doTransform"/>
  <to uri="activemq:Output"/>
</route>
```

You can also use Templating to consume a message from one destination, transform it with something like Velocity or XQuery and then send it on to another destination. For example using InOnly (one way messaging)

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

If you want to use InOut (request-reply) semantics to process requests on the **My.Queue** queue on ActiveMQ with a template generated response, then sending responses back to the JMSReplyTo Destination you could use this.

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

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

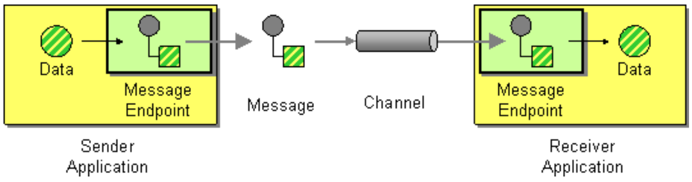
- TransformTest
- TransformViaDSLTest
- TransformProcessorTest
- TransformWithExpressionTest (test resource)

Using This Pattern

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

Message Endpoint

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



When using the DSL to create Routes you typically refer to Message Endpoints by their URIs rather than directly using the Endpoint interface. Its then a responsibility of the

CamelContext to create and activate the necessary Endpoint instances using the available Component implementations.

For more details see

- Message

Using This Pattern

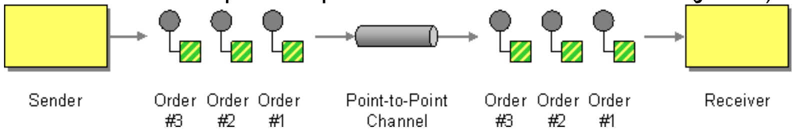
If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

MESSAGING CHANNELS

Point to Point Channel

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

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



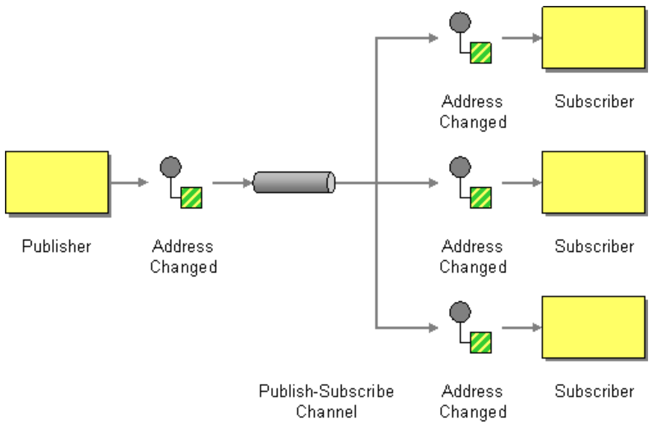
Using This Pattern

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

Publish Subscribe Channel

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

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



Using Routing Logic

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

Using the Fluent Builders

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

Using the Spring XML Extensions

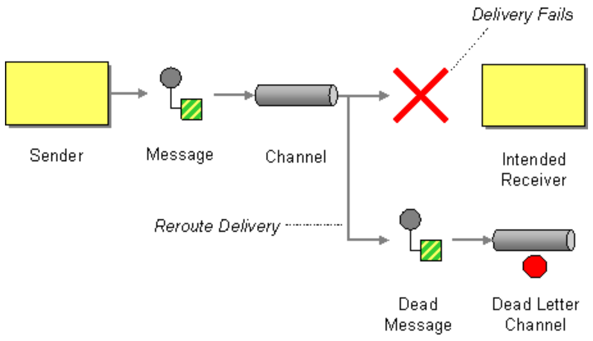
```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:b"/>
    <to uri="seda:c"/>
    <to uri="seda:d"/>
  </route>
</camelContext>
```

Using This Pattern

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

Dead Letter Channel

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



Redelivery

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

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

- how many times a message is attempted to be redelivered before it is considered a failure and sent to the dead letter channel
- the initial redelivery timeout
- whether or not exponential backoff is used (i.e. the time between retries increases using a backoff multiplier)
- whether to use collision avoidance to add some randomness to the timings

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

Redelivery default values

The default redeliver policy will use the following values:

- maximumRedeliveries=6
- initialRedeliveryDelay=1000L (1 second)
- maximumRedeliveryDelay = 60 * 1000L (60 seconds, **new option in Camel 1.4**)
- And the exponential backoff and collision avoidance is turned off.

The maximum redeliver delay ensures that a delay is never longer than the value, default 1 minute. This can happen if you turn on the exponential backoff.

Redelivery header

When a message is redelivered the DeadLetterChannel will append a customizable header to the message to indicate how many times its been redelivered. The default value is **org.apache.camel.redeliveryCount**.
The header **org.apache.camel.Redelivered** contains a boolean if the message is redelivered or not.

Configuring via the DSL

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

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("seda:errors"));
        from("seda:a").to("seda:b");
    }
};
```

You can also configure the RedeliveryPolicy as this example shows

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

errorHandler(deadLetterChannel("seda:errors").maximumRedeliveries(2).useExponentialBackOff())
        from("seda:a").to("seda:b");
    }
};
```

ExceptionPolicyStrategy (New feature in Camel 1.4)

ExceptionPolicyStrategy is a strategy for resolving which rule (ExceptionType) should handle the given thrown exception.

DeadLetterChannel supports pluggable strategies for resolving how exceptions should be handled. It is common to how different strategies for different types of exceptions. For instance network and IO related exceptions is more prone for network outages so the redeliver policy could have a higher attempts, timeouts etc. Where as a NullPointerException is typically a programming error so these kind of exception is severe and should not be redelivered. Camel uses a default strategy DefaultExceptionPolicyStrategy that applies the following rules:

- The exception type must be configured with an Exception that is an instance of the thrown exception
- If the exception type has exactly the thrown exception then its selected
- Otherwise the type that has an exception that is super of the thrown exception is selected (recurring up the exception hierarchy)

The example below illustrates a common exception handling configuration in Camel:

```
exception(NullPointerException.class)
    .maximumRedeliveries(1)
    .setHeader(MESSAGE_INFO, "Damm a NPE")
    .to(ERROR_QUEUE);

exception(IOException.class)
    .initialRedeliveryDelay(5000L)
    .maximumRedeliveries(3)
    .backOffMultiplier(1.0)
    .useExponentialBackOff()
    .setHeader(MESSAGE_INFO, "Damm somekind of IO exception")
    .to(ERROR_QUEUE);

exception(Exception.class)
    .initialRedeliveryDelay(1000L)
    .maximumRedeliveries(2)
    .setHeader(MESSAGE_INFO, "Damm just exception")
    .to(ERROR_QUEUE);
```

Here we have configured the handling of exceptions into three categories:

- NullPointerException (for special handling of these hard to track down bugs)
- IOException (for IO and network related issues we can attempt many times)
- Exception (fallback exception handling for all other kind of exceptions)

Camel will with the default strategy try to select the best suited category from above for any thrown exception.
So if a java.net.ScoketException is thrown then the IOException category will handle it. If a NumberFormatException or CamelExchangeException is thrown it is handled by the general purpose Exception category.

Camel supports pluggable exception policy strategies. See Error Handler for such details.

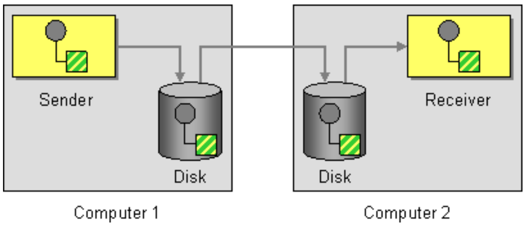
Using This Pattern

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

Guaranteed Delivery

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

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

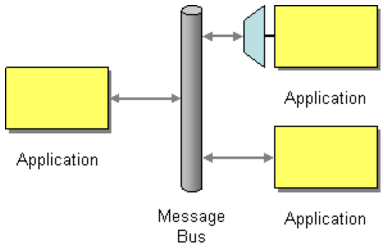


Using This Pattern

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

Message Bus

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



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

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

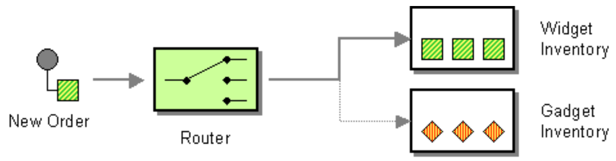
Using This Pattern

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

MESSAGE ROUTING

Content Based Router

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



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

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").choice().when(header("foo").isEqualTo("bar")).to("seda:b")

.when(header("foo").isEqualTo("cheese")).to("seda:c").otherwise().to("seda:d");
    }
};
```

Using the Spring XML Extensions

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <choice>
      <when>
        <xpath>$foo = 'bar'</xpath>
        <to uri="seda:b"/>
      </when>
      <when>
        <xpath>$foo = 'cheese'</xpath>
        <to uri="seda:c"/>
      </when>
      <otherwise>
        <to uri="seda:d"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

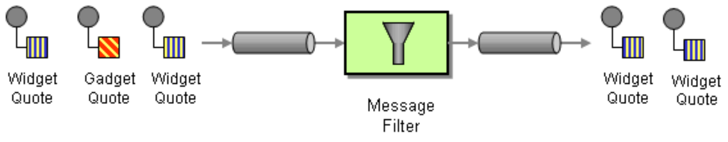
For further examples of this pattern in use you could look at the junit test case

Using This Pattern

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

Message Filter

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



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

Using the Fluent Builders

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

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

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

Using the Spring XML Extensions

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <filter>
      <xpath>$foo = 'bar'</xpath>
      <to uri="seda:b"/>
    </filter>
  </route>
</camelContext>
```

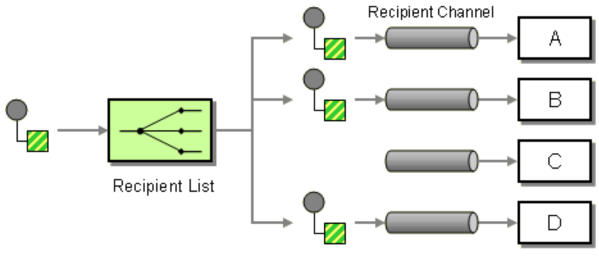
For further examples of this pattern in use you could look at the junit test case

Using This Pattern

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

Recipient List

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



Static Recipient List

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

Using the Fluent Builders

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

Using the Spring XML Extensions

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:b"/>
    <to uri="seda:c"/>
    <to uri="seda:d"/>
  </route>
</camelContext>
```

Dynamic Recipient List

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

Using the Fluent Builders

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

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

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

Using the Spring XML Extensions

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <recipientList>
      <xpath>$foo</xpath>
    </recipientList>
  </route>
</camelContext>
```

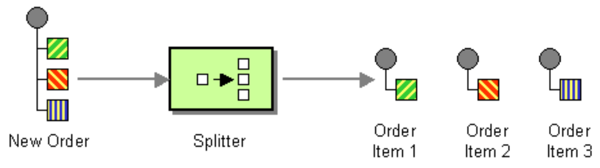
For further examples of this pattern in use you could look at one of the junit test case

Using This Pattern

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

Splitter

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



Example

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

Using the Fluent Builders

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

from("seda:a").splitter(bodyAs(String.class).tokenize("\n")).to("seda:b");
    }
};
```

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

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

Using the Spring XML Extensions

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <splitter>
      <xpath>/invoice/lineItems</xpath>
      <to uri="seda:b"/>
    </splitter>
  </route>
</camelContext>
```

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

Parallel execution of distinct 'parts'

If you want to execute all parts in parallel you can use special notation of `splitter()` with two arguments, where the second one is a **boolean** flag if processing should be parallel. e.g.

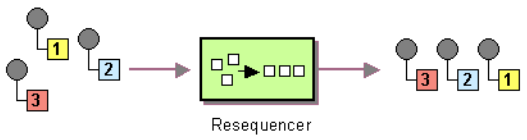
```
XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");
from("activemq:my.queue").splitter(xPathBuilder, true).to("activemq:my.parts");
```

Using This Pattern

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

Resequencer

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



Camel supports two resequencing algorithms:

- **Batch resequencing** collects messages into a batch, sorts the messages and sends them to their output.
- **Stream resequencing** re-orders (continuous) message streams based on the detection of gaps between messages.

Batch Resequencing

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

Using the Fluent Builders

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

This is equvalent to

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

To define a custom configuration for the batch-processing resequencer you should provide a configuration object.

```
from("direct:start").resequencer(body()).batch(new BatchResequencerConfig(300,
4000L)).to("mock:result")
```

This sets the batchSize to 300 and the batchTimeout to 4000 ms (by default, the batch size is 100 and the timeout is 1000 ms).

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

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

for example to reorder messages using their JMS priority.

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

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

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

Using the Spring XML Extensions

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start" />
    <resequencer>
      <simple>body</simple>
      <to uri="mock:result" />
      <!--
        batch-config can be ommitted for default (batch) resequencer settings
      -->
      <batch-config batchSize="300" batchTimeout="4000" />
    </resequencer>
  </route>
</camelContext>
```

Stream Resequencing

The next example shows how to use the stream-processing resequencer. Messages are re-ordered based on their sequence numbers given by a `seqnum` header using gap detection and timeouts on the level of individual messages.

Using the Fluent Builders

```
from("direct:start").resequencer(header("seqnum")).stream().to("mock:result");
```

To define a custom configuration for the stream-processing resequencer you should provide a configuration object.

```
from("direct:start").resequencer(header("seqnum")).stream(new
StreamResequencerConfig(5000, 4000L)).to("mock:result")
```

This sets the resequencer's capacity to 5000 and the timeout to 4000 ms (by default, the capacity is 100 and the timeout is 1000 ms).

The stream-processing resequencer algorithm is based on the detection of gaps in a message stream rather than on a fixed batch size. Gap detection in combination with timeouts removes the constraint of having to know the number of messages of a sequence (i.e. the batch size) in advance. Messages must contain a unique sequence number for which a predecessor and a successor is known. For example a message with the sequence number 3 has a predecessor message with the sequence number 2 and a successor message with the sequence number 4. The message sequence 2,3,5 has a gap because the sucessor of 3 is missing. The resequencer therefore has to retain message 5 until message 4 arrives (or a timeout occurs).

If the maximum time difference between messages (with successor/predecessor relationship with respect to the sequence number) in a message stream is known, then the resequencer's timeout parameter should be set to this value. In this case it is guaranteed that all messages of a stream are delivered in correct order to the next processor. The lower the timeout value is compared to the out-of-sequence time difference the higher is the probability for out-of-sequence messages delivered by this resequencer. Large timeout values should be supported by sufficiently high capacity values. The capacity parameter is used to prevent the resequencer from running out of memory.

By default, the stream resequencer expects `long` sequence numbers but other sequence numbers types can be supported as well by providing custom comparators.

```
ExpressionResultComparator<Exchange> comparator = new MyComparator();
StreamResequencerConfig config = new StreamResequencerConfig(5000, 4000L,
comparator);
from("direct:start").resequencer(header("seqnum")).stream(config).to("mock:result");
```

Using the Spring XML Extensions

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <resequencer>
      <simple>in.header.seqnum</simple>
      <to uri="mock:result" />
      <stream-config capacity="5000" timeout="4000"/>
    </resequencer>
  </route>
</camelContext>
```

Further Examples

For further examples of this pattern in use you could look at the batch-processing resequencer junit test case and the stream-processing resequencer junit test case

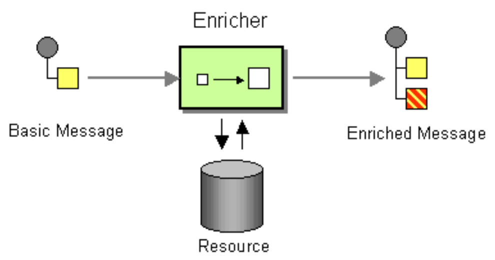
Using This Pattern

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

MESSAGE TRANSFORMATION

Content Enricher

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



Using the Fluent Builders

You can use Templating to consume a message from one destination, transform it with something like Velocity or XQuery and then send it on to another destination. For example using InOnly (one way messaging)

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

If you want to use InOut (request-reply) semantics to process requests on the **My.Queue** queue on ActiveMQ with a template generated response, then sending responses back to the JMSReplyTo Destination you could use this.

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

Here is a simple example using the DSL directly to transform the message body

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

In this example we add our own Processor using explicit Java code

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

Finally we can use Bean Integration to use any Java method on any bean to act as the transformer

```
from("activemq:My.Queue").
  beanRef("myBeanName", "myMethodName").
  to("activemq:Another.Queue");
```

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

- TransformTest
- TransformViaDSLTest

Using Spring XML

```
<route>
  <from uri="activemq:Input"/>
  <bean ref="myBeanName" method="doTransform"/>
  <to uri="activemq:Output"/>
</route>
```

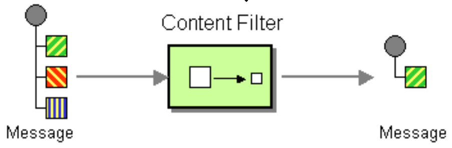
Using This Pattern

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

Content Filter

Camel supports the Content Filter from the EIP patterns using one of the following mechanisms in the routing logic to transform content from the inbound message.

- Message Translator
- invoking a Java bean
- Processor object



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

Using the Fluent Builders

Here is a simple example using the DSL directly

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

In this example we add our own Processor

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

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

- TransformTest
- TransformViaDSLTest

Using Spring XML

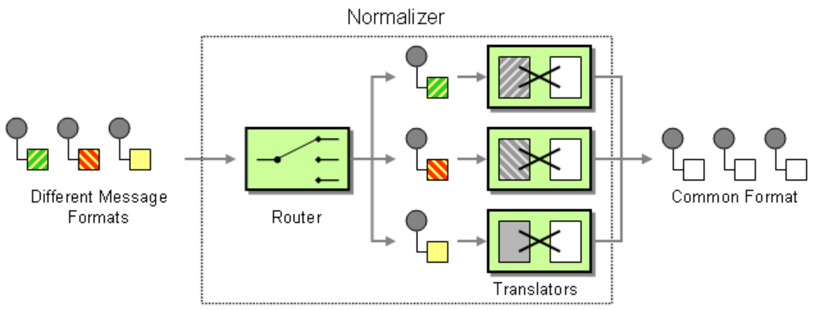
```
<route>
  <from uri="activemq:Input"/>
  <bean ref="myBeanName" method="doTransform"/>
  <to uri="activemq:Output"/>
</route>
```

Using This Pattern

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

Normalizer

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



See Also

- Message Router
- Content Based Router
- Message Translator

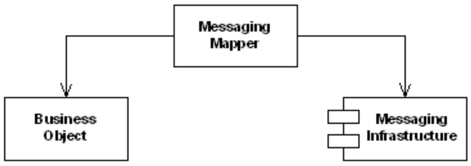
Using This Pattern

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

MESSAGING ENDPOINTS

Messaging Mapper

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



See also

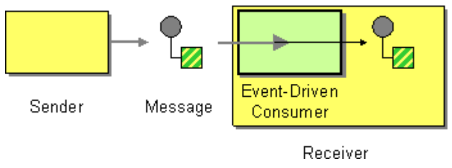
- Message Translator
- Type Converter
- CXF for JAX-WS support for binding business logic to messaging & web services
- POJO
- Bean

Using This Pattern

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

Event Driven Consumer

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



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

For more details see

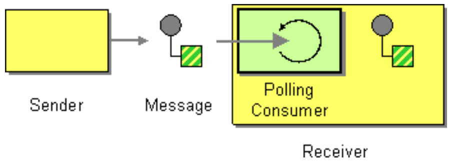
- Message
- Message Endpoint

Using This Pattern

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

Polling Consumer

Camel supports implementing the Polling Consumer from the EIP patterns using the PollingConsumer interface which can be created via the Endpoint.createPollingConsumer() method.



So in your Java code you can do

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

There are 3 main polling methods on PollingConsumer

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

Scheduled Poll Components

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

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

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

For more details see

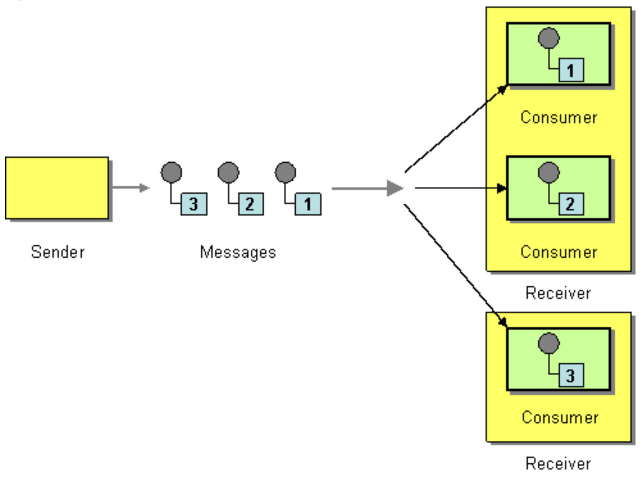
- PollingConsumer
- Scheduled Polling Components
    - ScheduledPollConsumer
    - File
    - JPA
    - Mail
    - Quartz

Using This Pattern

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

Competing Consumers

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



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

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

Enabling Competing Consumers with JMS

To enable Competing Consumers you just need to set the **concurrentConsumers** property on the JMS endpoint.

For example

```
from("jms:MyQueue?concurrentConsumers=5").bean(SomeBean.class);
```

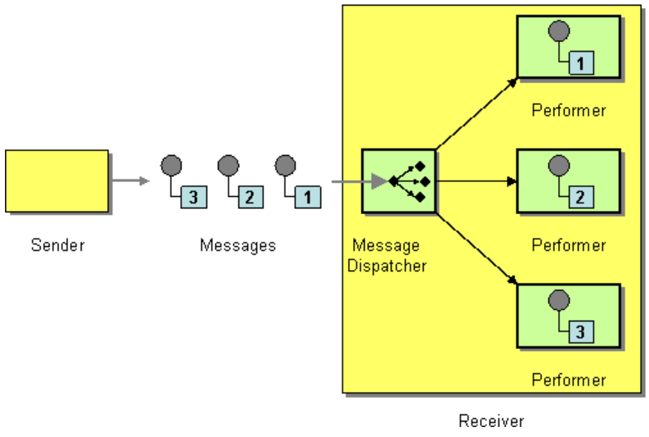
Or just run multiple JVMs of any ActiveMQ or JMS route

Using This Pattern

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

Message Dispatcher

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



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

See Also

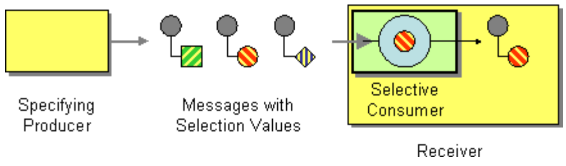
- JMS
- Selective Consumer
- Content Based Router
- Endpoint

Using This Pattern

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

Selective Consumer

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



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

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

Using the Fluent Builders

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

from("seda:a").filter(header("foo").isEqualTo("bar")).process(myProcessor);
    }
};
```

Using the Spring XML Extensions

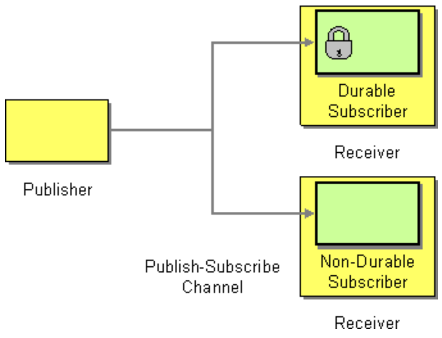
```
<bean id="myProcessor" class="org.apache.camel.builder.MyProcessor"/>

<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <filter>
      <xpath>$foo = 'bar'</xpath>
      <process ref="myProcessor"/>
    </filter>
  </route>
</camelContext>
```

Using This Pattern

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

Durable Subscriber

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



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

See Also

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

Using This Pattern

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

Idempotent Consumer

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

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

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

Using the Fluent Builders

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

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

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

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

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

Using the Spring XML Extensions

```
<bean id="myProcessor" class="org.apache.camel.builder.MyProcessor"/>

<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <filter>
      <xpath>$foo = 'bar'</xpath>
      <process ref="myProcessor"/>
    </filter>
  </route>
</camelContext>
```

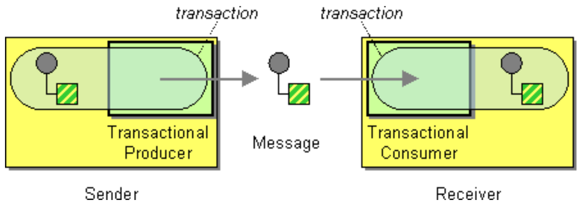
For further examples of this pattern in use you could look at the junit test case

Using This Pattern

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

Transactional Client

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



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

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

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

You first define needed object in the spring configuration.

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

  <bean id="jmsConnectionFactory"
class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
  </bean>
```

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

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

Transaction Policies

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

```
<bean id="PROPAGATION_REQUIRED"
class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="jmsTransactionManager"/>
  </bean>

  <bean id="PROPAGATION_NOT_SUPPORTED"
class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="jmsTransactionManager"/>
    <property name="propagationBehaviorName" value="PROPAGATION_NOT_SUPPORTED"/>
  </bean>

  <bean id="PROPAGATION_REQUIRES_NEW"
class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="jmsTransactionManager"/>
    <property name="propagationBehaviorName" value="PROPAGATION_REQUIRES_NEW"/>
  </bean>
```

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

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

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

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

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

Transaction Policies improvements in Camel 1.4

In Camel 1.4 we have eased the syntax to setup the transaction polices directly on the `SpringTransactionPolicy` object:

```
<bean id="PROPAGATION_REQUIRED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="jmsTransactionManager"/>
  </bean>

  <bean id="PROPAGATION_REQUIRES_NEW"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="jmsTransactionManager"/>
    <property name="propagationBehaviorName" value="PROPAGATION_REQUIRES_NEW"/>
  </bean>
```

And the Java DSL is a bit simpler now:

```
Policy requried = bean(SpringTransactionPolicy.class, "PROPAGATION_REQUIRED"));
```

Database Sample

In this sample we want to ensure that two endpoints is under transaction control. These two endpoints inserts data into a database.
The sample is in its full as a unit test.

First of all we setup the usual spring stuff in its configuration file. Here we have defined a DataSource to the HSQLDB and a most importantly
the Spring DataSoruce TransactionManager that is doing the heavy lifting of ensuring our transactional policies. You are of course free to use any
of the Spring based TransactionMananger, eg. if you are in a full blown J2EE container you could use JTA or the WebLogic or WebSphere specific managers.

We use the required transaction policy that we define as the `PROPOGATION_REQUIRED` spring bean. And as last we have our book service bean that does the business logic
and inserts data in the database as our core business logic.

```
<!-- datasource to the database -->
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:mem:camel"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
</bean>

<!-- spring transaction manager -->
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

```
<!-- policy for required transaction used in our Camel routes -->
<bean id="PROPAGATION_REQUIRED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager"/>
</bean>

<!-- bean for book business logic -->
<bean id="bookService" class="org.apache.camel.spring.interceptor.BookService">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

In our Camel route that is Java DSL based we setup the transactional policy, wrapped as a Policy.

```
// Notice that we use the SpringRouteBuilder that has a few more features than
// the standard RouteBuilder
return new SpringRouteBuilder() {
    public void configure() throws Exception {
        // setup the transaction policy
        SpringTransactionPolicy required = context.getRegistry()
            .lookup("PROPAGATION_REQUIRED", SpringTransactionPolicy.class);

        // use this error handler instead of DeadLetterChannel that is the
default
        // Notice: transactionErrorHandler is in SpringRouteBuilder
        if (useTransactionErrorHandler) {
            // useTransactionErrorHandler is only used for unit testing to reuse
code
            // for doing a 2nd test without this transaction error handler, so
ignore
            // this. For spring based transaction, end users is encured to use
the
            // transaction error handler instead of the default
DeadLetterChannel.
            errorHandler(transactionErrorHandler(required).
                // notice that the builder has builder methods for chained
configuration
                maximumRedeliveries(3).
                initialRedeliveryDelay(5 * 1000L));
        }
```

Then we are ready to define our Camel routes. We have two routes: 1 for success conditions, and 1 for a forced rollback condition.
This is after all based on a unit test.

```
// set the required policy for this route
from("direct:okay").policy(required).
    setBody(constant("Tiger in Action")).beanRef("bookService").
    setBody(constant("Elephant in Action")).beanRef("bookService");

// set the required policy for this route
from("direct:fail").policy(required).
```

```
    setBody(constant("Tiger in Action")).beanRef("bookService").
    setBody(constant("Donkey in Action")).beanRef("bookService");
```

As its a unit test we need to setup the database and this is easily done with Spring JdbcTemplate

```
// create database and insert dummy data
final DataSource ds = getMandatoryBean(DataSource.class, "dataSource");
jdbc = new JdbcTemplate(ds);
jdbc.execute("create table books (title varchar(50))");
jdbc.update("insert into books (title) values (?)", new Object[] {"Camel in
Action"});
```

And our core business service, the book service, will accept any books except the Donkeys.

```
public class BookService {

    private SimpleJdbcTemplate jdbc;

    public BookService() {
    }

    public void setDataSource(DataSource ds) {
        jdbc = new SimpleJdbcTemplate(ds);
    }

    public void orderBook(String title) throws Exception {
        if (title.startsWith("Donkey")) {
            throw new IllegalArgumentException("We don't have Donkeys, only
Camels");
        }

        // create new local datasource to store in DB
        jdbc.update("insert into books (title) values (?)", title);
    }
}
```

Then we are ready to fire the tests. First to commit condition:

```
public void testTransactionSuccess() throws Exception {
    template.sendBody("direct:okay", "Hello World");

    int count = jdbc.queryForInt("select count(*) from books");
    assertEquals("Number of books", 3, count);
}
```

And lastly the rollback condition since the 2nd book is a Donkey book:

```
public void testTransactionRollback() throws Exception {
    template.sendBody("direct:fail", "Hello World");

    int count = jdbc.queryForInt("select count(*) from books");
    assertEquals("Number of books", 1, count);
}
```

JMS Sample

In this sample we want to listen for messages on a queue and process the messages with our business logic java code and send them along. Since its based on a unit test the destination is a mock endpoint.

This time we want to setup the camel context and routes using the Spring XML syntax.

```
<!-- here we define our camel context -->
<camel:camelContext id="myroutes">
    <!-- and now our route using the XML syntax -->
    <camel:route>
        <!-- 1: from the jms queue -->
        <camel:from uri="activemq:queue:okay"/>
        <!-- 2: setup the transactional boundaries to require a transaction -->
        <camel:policy ref="PROPAGATION_REQUIRED"/>
        <!-- 3: call our business logic that is myProcessor -->
        <camel:process ref="myProcessor"/>
        <!-- 4: if success then send it to the mock -->
        <camel:to uri="mock:result"/>
    </camel:route>
</camel:camelContext>

<!-- this bean is our business logic -->
<bean id="myProcessor"
class="org.apache.camel.component.jms.tx.JMSTransactionalClientTest$MyProcessor"/>
```

Since the rest is standard XML stuff its nothing fancy now for the reader:

```
<bean id="jmsConnectionFactory"
class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
value="vm://localhost?broker.persistent=false&amp;broker.useJmx=false"/>
</bean>

<bean id="jmsTransactionManager"
class="org.springframework.jms.connection.JmsTransactionManager">
    <property name="connectionFactory" ref="jmsConnectionFactory"/>
</bean>

<bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
    <property name="connectionFactory" ref="jmsConnectionFactory"/>
    <property name="transactionManager" ref="jmsTransactionManager"/>
    <property name="transacted" value="true"/>
    <property name="concurrentConsumers" value="1"/>
</bean>

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

<bean id="PROPAGATION_REQUIRED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <constructor-arg>
```

```
        <bean
class="org.springframework.transaction.support.TransactionTemplate">
            <property name="transactionManager" ref="jmsTransactionManager"/>
        </bean>
    </constructor-arg>
</bean>
```

Our business logic is set to handle the incomming messages and fail the first two times. When its a success it responds with a `Bye World` message.

```
public static class MyProcessor implements Processor {
    private int count;

    public void process(Exchange exchange) throws Exception {
        if (++count <= 2) {
            throw new IllegalArgumentException("Forced Exception number " +
count + ", please retry");
        }
        exchange.getIn().setBody("Bye World");
        exchange.getIn().setHeader("count", count);
    }
}
```

And our unit test is tested with this java code. Notice that we expect the `Bye World` message to be delivered at the 3rd attempt.

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedMessageCount(1);
mock.expectedBodiesReceived("Bye World");
// success at 3rd attempt
mock.message(0).header("count").isEqualTo(3);

template.sendBody("activemq:queue:okay", "Hello World");

mock.assertIsSatisfied();
```

Spring based configuration

In Camel 1.4 we have introduced the concept of configuration of the error handlers using spring XML configuration. The sample below demonstrates that you can configure transaction error handlers in Spring XML as spring beans. These can then be set as global, per route based or per policy based error handler. The latter has been demonstrated in the samples above. This sample is the database sample configured in Spring XML.

Notice that we have defnined two error handler, one per route. The first route uses the transaction error handler, and the 2nd uses no error handler at all.

```
<!-- here we define our camel context -->
<camel:camelContext id="myroutes">

    <!-- first route with transaction error handler -->
    <!-- here we refer to our transaction error handler we define in this Spring
```

```
XML file -->
    <!-- in this route the transactionErrorHandler is used -->
    <camel:route errorHandlerRef="transactionErrorHandler">
        <!-- 1: from the jms queue -->
        <camel:from uri="activemq:queue:okay"/>
        <!-- 2: setup the transactional boundaries to require a transaction -->
        <camel:policy ref="required"/>
        <!-- 3: call our business logic that is myProcessor -->
        <camel:process ref="myProcessor"/>
        <!-- 4: if success then send it to the mock -->
        <camel:to uri="mock:result"/>
    </camel:route>

    <!-- 2nd route with no error handling -->
    <!-- this route doens't use error handler, in fact the spring bean with id
noErrorHandler -->
    <camel:route errorHandlerRef="noErrorHandler">
        <camel:from uri="activemq:queue:bad"/>
        <camel:to uri="log:bad"/>
    </camel:route>

</camel:camelContext>
```

The following snippet is the Spring XML configuration to setup the error handlers in pure spring XML:

```
<!-- camel policy we refer to in our route -->
<bean id="required" class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="template" ref="PROPAGATION_REQUIRED"/>
</bean>

<!-- the standard spring transaction template for required -->
<bean id="PROPAGATION_REQUIRED"
class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="jmsTransactionManager"/>
</bean>

<!-- the transaction error handle we refer to from the route -->
<bean id="transactionErrorHandler"
class="org.apache.camel.spring.spi.TransactionErrorHandlerBuilder">
    <property name="transactionTemplate" ref="PROPAGATION_REQUIRED"/>
    <!-- here we refer to the configurations of the error handler -->
    <property name="redeliveryPolicy" ref="redeliveryPolicyConfig"/>
</bean>

<!-- configuration of the transaction error handler -->
<bean id="redeliveryPolicyConfig"
class="org.apache.camel.processor.RedeliveryPolicy">
    <!-- try up till 5 times -->
    <property name="maximumRedeliveries" value="5"/>
    <!-- wait 5 seconds at first redelivery -->
    <property name="initialRedeliveryDelay" value="5000"/>
    <!-- increas the wait time for each redelivery -->
    <property name="useExponentialBackOff" value="true"/>
    <!-- wait at most 30 seconds between redelivery -->
    <property name="maximumRedeliveryDelay" value="30000"/>
</bean>

<!-- the no error handler -->
<bean id="noErrorHandler"
class="org.apache.camel.builder.NoErrorHandlerBuilder"/>
```

See Also

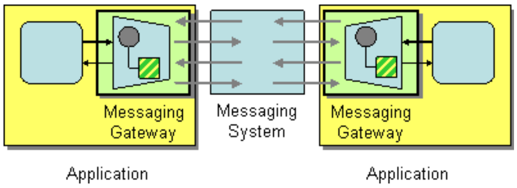
- JMS
- Dead Letter Channel
- Error Handler

Using This Pattern

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

Messaging Gateway

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



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

See Also

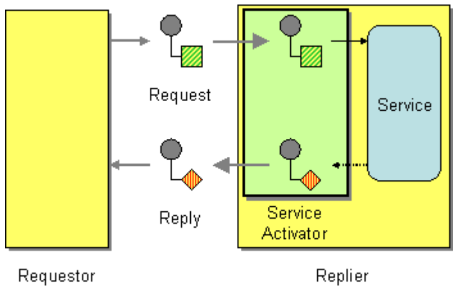
- Bean
- Pojo
- CXF

Using This Pattern

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

Service Activator

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



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

See Also

- Bean
- Pojo
- CXF

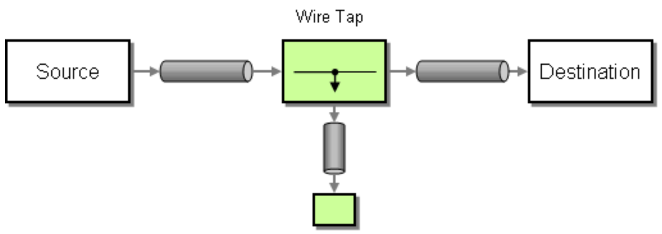
Using This Pattern

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

SYSTEM MANAGEMENT

Wire Tap

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



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

Using the Fluent Builders

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

Using the Spring XML Extensions

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:tap"/>
    <to uri="seda:b"/>
  </route>
</camelContext>
```

Using This Pattern

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

CHAPTER 6

° ° ° °

Component Appendix

There now follows the documentation on each Camel component.

ACTIVEMQ COMPONENT

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

This component is based on the JMS Component and uses Spring's JMS support for declarative transactions, using Spring's JmsTemplate for sending and a MessageListenerContainer for consuming.

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

URI format

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

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

```
activemq:FOO.BAR
```

You can be completely specific if you wish via

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

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

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

Configuring the Connection Factory

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

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

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.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://activemq.apache.org/camel/schema/spring http://activemq.apache.org/
camel/schema/spring/camel-spring.xsd">

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


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

</beans>
```

Invoking MessageListener POJOs in a Camel route

The ActiveMQ component also providers a helper Type Converter from a JMS MessageListener to a Processor. This means that the Bean component is capable of invoking any JMS MessageListener bean directly inside any route.

So for example you can create a MessageListener in JMS like this....

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

Then use it in your Camel route as follows

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

i.e. you can reuse any of the Camel Components and easily integrate them into your JMS MessageListener POJO!

Getting Component JAR

The ActiveMQ Camel component is released with the ActiveMQ project itself.

For Maven 2 users you simply just need to add the following dependency to your project.

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

Alternatively you can download the component jar directly from the Maven repository:
activemq-core-5.1.0.jar

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

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 to 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 1 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:directory-name[?options]
```

So for example to send to the journal located in the /tmp/data directory you would use

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

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 objects with the in message :

- header "journal" : set to the endpoint uri of the journal the message came from
- header "location" : set to a Location which identifies where the recored was stored on disk
- body : set to ByteSequence which contains the byte array data of the stored message

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

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

AMQP

The AMQP component supports the AMQP protocol via the Qpid project.

URI format

```
amqp:[queue:][topic:]destinationName[?option1=value[&option2=value2]]
```

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

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

ATOM COMPONENT

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

Camel will default poll the feed every 60th seconds.
Note: The component currently only supports polling (consuming) feeds.

URI format

```
atom://atomUri
```

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

Options

Property	Default	Description
splitEntries	true	If **true** Camel will poll the feed and for the subsequent polls return each entry poll by poll. If the feed contains 7 entries then Camel will return the first entry on the first poll, the 2nd entry on the next poll, until no more entries where as Camel will do a new update on the feed. If **false** then Camel will poll a fresh feed on every invocation.
filter	true	Is only used by the split entries to filter the entries to return. Camel 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`.
consumer.delay	60000	Delay in millis between each poll
consumer.initialDelay	1000	Millis before polling starts
consumer.userFixedDelay	false	**true** to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.

Exchange data format

Camel will set the in body on the returned Exchange with the entries. Depending on the splitEntries flag Camel 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 entires list of entries from the feed is set: `exchange.in.body(List<Entry>)`

Camel will set the Feed object on the in header:

```
exchange.in.header("org.apache.camel.component.atom.feed", feed)
```

Samples

In this sample we poll James Strahams 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 setup Camel standalone, not running in any Container or using Spring.

```
// This is the CamelContext that is the heart of Camel
private CamelContext context;

// We use a simple Hashtable for our bean registry. For more advanced usage
Spring is supported out-of-the-box
private Hashtable beans = new Hashtable();

// We iniitalize Camel
private void setupCamel() throws Exception {
    // First we register a blog service in our bean registry
    beans.put("blogService", new BlogService());

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

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

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

/**
 * This is the route builder where we create our routes in the advanced Camel
DSL syntax
 */
```

```
protected RouteBuilder createRouteBuilder() 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&consumer.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",
"goodBlog").to("seda:goodBlogs");

            // 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
 */
public void testFiltering() throws Exception {
    // Get the mock endpoint
    MockEndpoint mock = context.getEndpoint("mock:result", MockEndpoint.class);

    // There should be two good blog entries from the feed
    mock.expectedMessageCount(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();
    }

    /**
     * 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;
        }

    }
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

BEAN COMPONENT

The **bean:** component binds beans to Camel message exchanges.

URI format

```
bean:someName[?methodName=someMethod]
```

Where **someName** can be any string which is used to lookup the bean in the Registry and **someMethod** defines the name of the method to invoke.

This will use the Bean Binding to map the message exchange to the bean.

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.xml; or if you don't use Spring then 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 Camel routes that use it to process exchanges.

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

A **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 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);
```

Bean binding

The binding of a Camel Message to a bean method call can occur in different ways

- if the bean can be converted to a Processor using the Type Converter mechanism then this is used to process the message. This mechanism is used by the ActiveMQ component to allow any MessageListener to be invoked by the Bean component
- if the body of the message can be converted to a BeanInvocation (the default payload used by the ProxyHelper) - then that its used to invoke the method and pass the arguments
- if the message contains the header **org.apache.camel.MethodName** then that method is invoked, converting the body to whatever the argument is to the method
- otherwise the type of the method body is used to try find a method which matches; an error is thrown if a single method cannot be chosen unambiguously.
- you can also use Exchange as the parameter itself, but then the return type must be void.

By default the return value is set on the outbound message body.

For example a POJO such as:

```
public class Bar {
    public String doSomething(String body) {
      // process the in body and return whatever you want
      return "Bye World";
   }
```

Or the Exchange example. Notice that the return type must be **void**:

```
public class Bar {

    public void doSomething(Exchange exchange) {
      // process the exchange
      exchange.getIn().setBody("Bye World");
   }
```

For example you could write a method like this (showing also a feature in Camel, the @MessageDrive annotation):

```
public class Foo {

    @MessageDriven(uri = "activemq:my.queue")
    public void doSomething(String body) {
        // process the inbound message here
    }

}
```

Here Camel with subscribe to an ActiveMQ queue, then convert the message payload to a String (so dealing with TextMessage, ObjectMessage and BytesMessage in JMS), then process this method.

Using Annotations to bind parameters to the Exchange

The annotations can be used to bind in situations where traditional methods would result in ambiguous methods. So by adding annotations you can decorate your bean to help Camel invoke the correct method.

You can also use the following annotations to bind parameters to different kinds of Expression

Annotation	Meaning
@Body	To bind to an inbound message body
@Header	To bind to an inbound message header
@Headers	To bind to the Map of the inbound message headers
@OutHeader	To bind to an outbound message header
@OutHeaders	To bind to the Map of the outbound message headers
@Property	To bind to a named property on the exchange

@Properties	To bind to the property map on the exchange

For example

```
public class Foo {

    @MessageDriven(uri = "activemq:my.queue")
    public void doSomething(@Header('JMSCorrelationID') String correlationID,
@Body String body) {
        // process the inbound message here
    }

}
```

In the above you can now pass the Message.getJMSCorrelationID() as a parameter to the method (using the Type Converter to adapt the value to the type of the parameter).

Finally you don't need the @MessageDriven annotation; as the Camel route could describe which method to invoke.

e.g. a route could look like

```
from("activemq:someQueue").
  to("bean:myBean");
```

Here **myBean** would be looked up in the Registry (such as JNDI or the Spring ApplicationContext), then the body of the message would be used to try figure out what method to call.

If you want to be explicit you can use

```
from("activemq:someQueue").
  to("bean:myBean?methodName=doSomething");
```

And here we have a nifty example for you to show some great power in Camel. You can mix and match the annotations with the normal parameters, so we can have this example with annotations and the Exchange also:

```
public void doSomething(@Header(name = "user") String user, @Body String body,
Exchange exchange) {
        exchange.getIn().setBody(body + "MyBean");
    }
```

Using Expression Languages

You can also use any of the Languages supported in Camel to bind expressions to method parameters when using bean integration. For example you can use any of these annotations...

Annotation	Description
@BeanShell	Inject a BeanShell expression
@EL	Inject an EL expression

@Groovy	Inject a Groovy expression
@JavaScript	Inject a JavaScript expression
@OGNL	Inject an OGNL expression
@PHP	Inject a PHP expression
@Python	Inject a Python expression
@Ruby	Inject a Ruby expression
@Simple	Inject an Simple expression
@XPath	Inject an XPath expression
@XQuery	Inject an XQuery expression

For example

```
public class Foo {

    @MessageDriven(uri = "activemq:my.queue")
    public void doSomething(@Path("/foo/bar/text()") String correlationID, @Body
String body) {
        // process the inbound message here
    }
}
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

CXF COMPONENT

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

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>
```

URI format

```
cxf://address?options
```

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

```
cxf:bean:cxfEndpoint
```

Where **cxfEndpoint** represents the spring bean's name which presents the CXF endpoint
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	Example	Required?	default value
wsdlURL	The location of the WSDL.	file://local/wsdl/ hello.wsdl or wsdl/ hello.wsdl	No	WSDL is obtained from endpoint address by default
serviceClass	The name of the SEI(Service Endpoint Interface) implementation class. This class can have but does not require JSR181 annotations.	org.apache.camel.Hello	Yes for CXF provider, for CXF Consumer only if POJO dataFormat option is used	Ê
serviceName	The service name this service is implementing, it maps to the wsdl:service@name.	{http://org.apache.camel} ServiceName	Only if more than one serviceName in WSDL present	Ê

portName	The port name this service is implementing, it maps to the wsdl:port@name.	{http://org.apache.camel} PortName	Only if more than one portName under the serviceName is present	Ê
dataFormat	Which data type messages the CXF endpoint supports	POJO, PAYLOAD, MESSAGE	No	POJO
wrapped	Which kind of operation that CXF endpoint producer will invoke	true, false	No	false
setDefaultBus	Will set the default bus when CXF endpoint create a bus by itself	true, false	No	false

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

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.
PAYLOAD	PAYLOAD is the message payload (the contents of the soap:body) after message configuration in the CXF endpoint is applied.
MESSAGE	MESSAGE is the raw message that is received from the transport layer.

Configure the CXF endpoints with spring

You can configure the CXF endpoint with the below spring configuration file, and you can also embed the endpoint into the camelContext tags.

```
<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.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://activemq.apache.org/camel/schema/cxfEndpoint
http://activemq.apache.org/camel/schema/cxf/cxfEndpoint.xsd
       http://activemq.apache.org/camel/schema/spring http://activemq.apache.org/
camel/schema/spring/camel-spring.xsd
    ">

   <cxf:cxfEndpoint id="routerEndpoint" address="http://localhost:9003/
CamelContext/RouterPort"
         serviceClass="org.apache.hello_world_soap_http.GreeterImpl"/>

   <cxf:cxfEndpoint id="serviceEndpoint" address="http://localhost:9000/
SoapContext/SoapPort"
         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>
```

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 jaxws 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>s or <ref>s.
cxf:inFaultInterceptors	The incoming fault interceptors for this endpoint. A list of <bean>s or <ref>s.
cxf:outInterceptors	The outgoing interceptors for this endpoint. A list of <bean>s or <ref>s.
cxf:outFaultInterceptors	The outgoing fault interceptors for this endpoint. A list of <bean>s or <ref>s.
cxf:properties	A properties map 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 and properties here:
http://cwiki.apache.org/CXF20DOC/jax-ws-configuration.html

NOTE

You can use cxf:properties to set the camel-cxf endpoint's dataFormat and setDefaultBus properties from spring configuration file.

```
<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 let camel-cxf component to use log4j instead of java.util.logging

CXF's default logger is using java.util.logging, if you want to change it to log4j.
Here is the instruction: 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 consume the message from the camel-cxf endpoint in POJO data format

The camel-cxf endpoint consumer POJO data format is based on the cxf invoker, 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);

    public void process(Exchange exchange) throws Exception {
        LOG.info("processing exchange in camel");
        // 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 exception to trigger it
            // We just set the fault message in the exchange for camel-cxf
component handling
            exchange.getFault().setBody(fault);
        }
        name.value = "Bonjour";
        ssn.value = "123";
        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. First you need to specify the operation name in the message header , then add the method parameters into a list and set the message with this parameter list will be ok. The response message's body is an object array, you can get the result from that array.

```
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 object array 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 array.
Object[] output = (Object[])out.getBody();
LOG.info("Received output text: " + output[0]);
Map<String, Object> responseContext =
CastUtils.cast((Map)out.getHeader(Client.RESPONSE_CONTEXT));
assertNotNull(responseContext);
assertEquals("We should get the response context here", "UTF-8",
responseContext.get(org.apache.cxf.message.Message.ENCODING));
assertEquals("Reply body on Camel is wrong", "echo " + TEST_MESSAGE, output[0]);
```

How to propagate camel-cxf endpoint's request and response context

cxf client API provides a way to invoke the operation with request and response context. If you are using camel-cxf endpoint producer to invoke the outside web service, you can set the request context and get response context with below codes.

```
CxfExchange exchange = (CxfExchange)template.send(getJaxwsEndpointUri(), new
Processor() {
            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_SERVER_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 fist element of the array is the
return value
        Object[] output = (Object[])out.getBody();
        LOG.info("Received output text: " + output[0]);
        // Get the response context form outMessage
        Map<String, Object> responseContext =
CastUtils.cast((Map)out.getHeader(Client.RESPONSE_CONTEXT));
        assertNotNull(responseContext);
        assertEquals("Get the wrong wsdl opertion name", "{http://apache.org/
hello_world_soap_http}greetMe",
responseContext.get("javax.xml.ws.wsdl.operation").toString());
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

DATASET COMPONENT

Testing of distributed and asynchronous processing is notoriously difficult. The Mock, Test and DataSet endpoints work great with the Spring Testing framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's large range of Components together with the powerful Mock and Test testing endpoints.
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 both as a source of messages and as a way to assert that the data set is received.

URI format

```
dataset:name
```

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

Example

For example to test that a set of messages are sent to a queue then consumed from a queue without loosing 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	Description
defaultBody	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	Specifies the number of messages to be received before reporting progress. Useful for showing progress of a large load test
size	Specifies how many messages to send/consume

Load testing ActiveMQ with Camel

There is an example of load testing an ActiveMQ queue using Camel in the ActiveMQ source code repository. The code lives at this location

- https://svn.apache.org/repos/asf/activemq/trunk/activemq-camel-loadtest/

You can grab the code

```
svn co https://svn.apache.org/repos/asf/activemq/trunk/activemq-camel-loadtest/
```

Then try running the test case

```
cd activemq-camel-loadtest
mvn clean install
```

To see how the test is defined see the Spring XML file

```
<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-2.0.xsd
       http://activemq.apache.org/camel/schema/spring http://activemq.apache.org/
camel/schema/spring/camel-spring.xsd
       http://activemq.apache.org/schema/core http://activemq.apache.org/schema/
core/activemq-core.xsd
    ">

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

    <route>
      <from uri="dataset:myDataSet"/>
      <to uri="activemq:test.queue"/>
    </route>

    <route>
      <from uri="activemq:test.queue"/>
      <to uri="dataset:myDataSet"/>
    </route>

  </camelContext>

  <bean id="myDataSet" class="org.apache.camel.component.dataset.SimpleDataSet">
    <property name="size" value="10000"/>
    <property name="reportCount" value="100"/>
  </bean>

  <!-- lets create an embedded broker for this test -->
  <broker xmlns="http://activemq.apache.org/schema/core" dataDirectory="target/
activemq">

    <transportConnectors>
       <transportConnector uri="tcp://localhost:61616"/>
    </transportConnectors>

  </broker>
</beans>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Spring Testing

DIRECT COMPONENT

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

URI format

```
direct:someName
```

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

Options

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

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

ESPER

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

URI format

```
esper:name[?option1=value[&option2=value2]]
```

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	Ê	The Esper Pattern expression as a String to filter events
eql	Ê	The Esper EQL expression as a String to filter events

Demo

There is a demo which shows how to work with ActiveMQ, Camel and Esper in the Camel Extra project

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Esper Camel Demo

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
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

FILE COMPONENT

The File component provides access to file systems; allowing files to be processed by any other Camel Components or messages from other components can be saved to disk.

URI format

```
file:fileOrDirectoryName
```

or

```
file://fileOrDirectoryName
```

Where **fileOrDirectoryName** represents the underlying file name. Camel will determine if **fileOrDirectoryName** is a file or directory.

URI Options

Name	Default Value	Description
initialDelay	1000	Camel 1.3 or older: milliseconds before polling the file/directory starts
delay	500	Camel 1.3 or older: milliseconds before the next poll of the file/directory
useFixedDelay	false	Camel 1.3 or older: **true** to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
consumer.initialDelay	1000	Camel 1.4: milliseconds before polling the file/directory starts
consumer.delay	500	Camel 1.4: milliseconds before the next poll of the file/directory
consumer.useFixedDelay	false	Camel 1.4: **true** to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
recursive	true	if a directory, will look for changes in files in all the sub directories

Important Information

See the section "Common gotchas with folder and filenames" below.

lock	true	if true will lock the file for the duration of the processing
regexPattern	null	will only fire a an exchange for a file that matches the regex pattern
delete	false	If delete is true then the file will be deleted when it is processed (the default is to move it, see below)
noop	false	If true then the file is not moved or deleted in any way (see below). This option is good for read only data, or for ETL type requirements
moveNamePrefix	null	The prefix String perpended to the filename when moving it. For example to move processed files into the *done* directory, set this value to 'done/'
moveNamePostfix	null	The postfix String appended to the filename when moving it. For example to rename processed files from *foo* to *foo.old* set this value to '.old'
append	true	When writing do we append to the end of the file, or replace it?
autoCreate	true	If set to true Camel will create the directory to the file if the file path does not exists - Uses File#mkdirs()
bufferSize	128kb	Write buffer sized in bytes. Camel uses a default of 128 * 1024 bytes.
ignoreFileNameHeader	false	If this flag is enabled then producers will ignore the 'org.apache.camel.file.name' header and generate a new dynamic filename

excludedNamePrefixes	"."	Is used to exclude files if filename is starting with any of the given prefixes. The parameter is a String[]
excludedNamePostfixes	".camelLock"	Is used to exclude files if filename is ending with any of the given prefixes. The parameter is a String[]
generateEmptyExchangeWhenIdle	false	Option only for the FileConsumer. If this option is **true** and there was no files to process we simulate processing a single empty file, so an exchange is happening. In this situation the file parameter in FileExchange is null.

By default the file is locked for the duration of the processing. Also when files are processed they are moved into the *.camel* directory; so that they appear to be deleted.

Message Headers

The following message headers can be used to affect the behavior of the component

Header	Description
org.apache.camel.file.name	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 then a generated message ID is used as filename instead.
org.apache.camel.file.name.produced	New in Camel 1.4: 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 the name of the file that was written.

Common gotchas with folder and filenames

When Camel is producing files (writing files) there are a few gotchas how to set a filename of your choice. By default Camel 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. Such a filename is not desired and therefore best practice is to provide the filename in the message header "org.apache.camel.file.name".

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(FileComponent.HEADER_FILE_NAME,
"report.txt").to( "file:target/reports");
```

Canel will default try to auto create the folder if it does not exists, and this is a bad combination with the UUID filename from above. So if you have:

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

And you want Camel to store in the file report.txt and autoCreate is true, then Camel will create the folder: target/reports/report.txt/. To fix this set the autoCreate=false and create the folder target/reports manually.

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

With auto create disabled Camel will store the report in the report.txt as expected.

File consumer, scanning for new files gotcha

The file consumer scans for new files by keeping an internal modified timestamp of the last consumed file. So if you copy a new file that has an older modified timestamp, then Camel will **not** pickup this file. This can happen if you are testing and you copy the same file back to the folder that has just been consumed. To remedy this modify the timestamp before copying the file back.

Samples

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.

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
  }
});
```

Body will be File object pointing to the file that was just dropped to 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 as 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 to do.

Writing to files

Camel is of course also able to write files, eg. producing 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 {
    template.sendBody("seda:reports", "This is a great report");

    // give time for the file to be written before assertions
    Thread.sleep(1000);

    // assert the file exists
    File file = new File("target/test-reports/report.txt");
    file = file.getAbsoluteFile();
    assertTrue("The file should have been written", file.exists());
}

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("seda:reports").processRef("processReport").to("file://target/
test-reports");
        }
    };
}

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(FileComponent.HEADER_FILE_NAME,
"report.txt");
    }

}
```

FileProducer filename gotchas

This unit test demonstrates some of the gotchas with filenames for the File Producer.

```
public void testProducerWithMessageIdAsFileName() throws Exception {
    Endpoint endpoint = context.getEndpoint("direct:report");
    Exchange exchange = endpoint.createExchange();
    exchange.getIn().setBody("This is a good report");

    FileEndpoint fileEndpoint = resolveMandatoryEndpoint("file:target/reports/
report.txt", FileEndpoint.class);
    String id = fileEndpoint.getGeneratedFileName(exchange.getIn());

    template.send("direct:report", exchange);

    File file = new File("target/reports/report.txt/" + id);
    assertEquals("File should exists", true, file.exists());
}

public void testProducerWithConfiguedFileNameInEndpointURI() throws Exception {
    template.sendBody("direct:report2", "This is another good report");
    File file = new File("target/report2.txt");
    assertEquals("File should exists", true, file.exists());
}

public void testProducerWithHeaderFileName() throws Exception {
    template.sendBody("direct:report3", "This is super good report");
    File file = new File("target/report-super.txt");
    assertEquals("File should exists", true, file.exists());
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:report").to("file:target/reports/report.txt");

            from("direct:report2").to("file:target/
report2.txt?autoCreate=false");

            from("direct:report3").setHeader(FileComponent.HEADER_FILE_NAME,
"report-super.txt").to("file:target/");
        }
    };
}
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

FIX

The FIX component supports the FIX protocol by using the QuickFix/J library.

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 refer to a full URL using http: or file: schemes.

Message Formats

By default this component will attempt to use the Type Converter to turn the inbound message body into a QuickFix Message class and all outputs from FIX will be in the same format.

If you are using the Artix Data Services support then any payload such as files or streams or byte arrays can be converted nicely into FIX messages.

Using camel-fix

To use this module you need to use the FUSE Mediation Router distribution. Or you could just add the following to your pom.xml, substituting the version number for the latest & greatest release.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-parent</artifactId>
  <version>1.3.0.1-fuse</version>
</dependency>
```

And ensure you are pointing at the maven repo

```
<repository>
     <id>open.iona.m2</id>
```

```
    <name>IONA Open Source Community Release Repository</name>
    <url>http://repo.open.iona.com/maven2</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </repository>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

FLATPACK COMPONENT

The Flatpack component supports fixed width and delimited file parsing via the FlatPack library.

URI format

```
flatpack:[delim|fixed]:flatPackConfig.pzmap.xml
```

or for a delimited file handler with no configuration file just use

```
flatpack:someName
```

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

URI Options

Name	Default Value	Description
delimiter	','	the default character delimiter for delimited files
textQualifier	""	the text qualifier delimited files

ignoreFirstRecord	true	whether the first line is ignored for delimited files (for the column headers)

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

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

FTP/SFTP COMPONENT

This component provides access to remote file systems over the FTP and SFTP protocols.
This component is an extension of the File component.

URI format

```
ftp://[username@]hostname[:port]/filename[?options]
sftp://[username@]hostname[:port]/filename[?options]
```

Where **filename** represents the underlying file name or directory. Can contain nested folders.
The **username** is currently only possible to provide in the hostname parameter.
If no **port** number is provided. Camel will provide default values according to the protocol. (ftp = 21, sftp = 22)

Examples

```
ftp://someone@someftpserver.com/public/upload/images/
holiday2008?password=secret&binary=true
ftp://someoneelse@someotherftpserver.co.uk:12049/reports/2008/
budget.txt?password=secret&binary=false&directory=false
ftp://publicftpserver.com/download
```

Options

Name	Default Value	Description
directory	true	indicates whether or not the given file name should be interpreted by default as a directory or file (as it sometimes hard to be sure with some FTP servers)
password	null	specifies the password to use to login to the remote file system
binary	false	specifies the file transfer mode BINARY or ASCII. Default is ASCII.
setNames	false	Used by FTPConsumer. If set to true Camel will set the special filename header FileComponent.HEADER_FILE_NAME value to the filename from the FTP Server. **Note:** In Camel 1.4 the default value has changed to **true**.
consumer.delay	500	Delay in millis between each poll
consumer.initialDelay	1000	Millis before polling starts
consumer.userFixedDelay	false	**true** to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.

As this component is an extension to the File component the options from this parent component is also available.

Message Headers

The following message headers is provided in the message.

Header	Description
file.remote.host	The hostname of the remote server

file.remote.name	The fullname of the file consumed from the remote server

Consumer properties

When using FTPConsumer (downloading files from a FTP Server) the consumer specific properties from the File component should be prefixed with "consumer.". For example the delay option from File Component should be specified as "consumer.delay=30000" in the URI. See the samples or some of the unit tests of this component.

Known issues

When consuming files (downloading) you must use type conversation to either String or to InputStream for ASCII and BINARY file types.
In Camel 1.4 this is fixed, as there are build in type converters for the ASCII and BINARY file types, meaning that you do not need the convertBodyTo expression.

Also in Camel 1.3 since setNames is default **false** then you must explicitly set the filename using the setHeader expression when consuming from FTP directly to File.
The code below illustrates this:

```
private String ftpUrl =
"ftp://camelrider@localhost:21/public/downloads?password=admin&binary=false";
private String fileUrl = "file:myfolder/?append=false&noop=true";

return new RouteBuilder() {
    public void configure() throws Exception {
        from(ftpUrl).setHeader(FileComponent.HEADER_FILE_NAME,
constant("downloaded.txt")).convertBodyTo(String.class).to(fileUrl);
    }
};
```

Or you can set the option to **true** as illustrated below:

```
private String ftpUrl =
"ftp://camelrider@localhost:21/public/downloads?password=admin&binary=false&setNames=true";
private String fileUrl = "file:myfolder/?append=false&noop=true";

return new RouteBuilder() {
    public void configure() throws Exception {
        from(ftpUrl).convertBodyTo(String.class).to(fileUrl);
    }
};
```

Sample

In the sample below we setup Camel 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. Camel 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 Component.

from("ftp://scott@localhost/public/reports?password=tiger&binary=true&consumer.delay="
+ delay).
                to("file://target/test-reports");
        }
    };
}
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

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 project which hosts all *GPL related components for Camel.

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 then 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 which will be invoked on your entity bean when the entity bean is consumed.

URI format

```
hibernate:[entityClassName]
```

For sending to the endpoint, the **entityClassName** is optional. If specified it is used to help use the [Type Conversion] to ensure the body is of the correct type.

For consuming the **entityClassName** is mandatory.

Options

Name	Default Value	Description
entityType	entityClassName	Is the provided entityClassName from the URI
consumeDelete	true	Option for HibernateConsumer only. Enables / disables whether or not the entity is deleted after it is consumed.
consumeLockEntity	true	Option for HibernateConsumer only. Enables / disables 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 after the entity beans has been persisted.
maximumResults	-1	Option for HibernateConsumer only. Set the maximum number of results to retrieve on the Query.
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. **true** to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

HTTP COMPONENT

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

URI format

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

Usage

You can only produce to endpoints generated by the HTTP component. Therefore it should never be used aas input into your camel Routes. To bind/expose an HTTP endpoint via an http server as input to a camel route, you can use the Jetty Component

How to set the POST/PUT/INFO/DELETE/GET 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;

```
new RouteBuilder() {
    public void configure() {
        from("direct:start")
            .setHeader(org.apache.camel.component.http.HttpMethods.HTTP_METHOD,
org.apache.camel.component.http.HttpMethods.POST)
        .to("http://www.google.com")
            .to("mock:results");
    }
};
```

And the equivalent spring sample:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <setHeader headerName="http.requestMethod" value="POST"/>
    <to uri="http://www.google.com"/>
    <to uri="mock:results"/>
  </route>
</camelContext>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

IBATIS

The **ibatis:** component allows you to query, poll, insert, update and delete data in a relational database using Apache iBATIS.

URI format

```
ibatis:operationName
```

Where **operationName** is the name in the iBATIS XML configuration file which maps to the query, insert, update or delete operation you wish to evaluate.

For example if you wish to poll a database for rows using iBATIS and then send them to a JMS Queue via ActiveMQ you could do

```
from("ibatis:selectAllAccounts").
    to("activemq:MyQueue");
```

Or to consume beans from a JMS queue and insert them into a database you could do...

```
from("activemq:Some.Queue").
  to("ibatis:insertAccount");
```

Options

Name	Default Value	Description

initialDelay	1000	The number of milliseconds until the first poll when polling (consuming)
delay	500	The number of milliseconds for the subsequent delays after the intial delay
useFixedDelay	false	Whether or not the fixed delay is to be used, to enable a repeated timer

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

|

IRC COMPONENT

The **irc:** component implements an IRC (Iternet Relay Chat) transport.

URI format

```
irc:host[:port]/#room
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JAVASPACE COMPONENT

The **javaspace:** component is a transport for working with any JavaSpace compliant implementation, this component has been tested with both the Blitz implementation and the GigaSpace implementation .
This component can be used for sending and receiving any object inheriting from the Jini Entry class, it's also possible to pass an id (Spring Bean) of a template that can be used for reading/ taking the entries from the space.
This component can be also 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, then, for invoking remotely a POJO using as a transport a JavaSpace.
This latter feature can be used for an easy implementation of the master/worker pattern where a POJO provides the business logic for the worker.
Look at the test cases for seeing the various usage option for this component.

URI format

```
javaspace:jini://host?options
```

Examples

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
```

In this case the payload can be any object inheriting from the Jini Entry.

Sending and receiving serializable objects

Using the routes as above it's also possible to send and receive any serializable object. The camel component detects that the payload is not a Jini Entry and then it automatically wraps the payload into a Camel Jini Entry. In this way a JavaSpace can be used as a generic transport.

Using JavaSpace as a remote invocation transport

The camel-javaspace component has been tailored to work in combination with the camel-bean component. It's possible, then, to call a remote POJO using JavaSpace as a transport:

```
from("direct:input").to("javaspace:jini://localhost?spaceName=mySpace");
//Client side

from("javaspace:jini://localhost?concurrentConsumers=10&spaceName=mySpace")
//Server side
```

In the code there are two test cases showing how to use the a POJO for realizing the master/worker pattern. The idea is to use the POJO for providing the business logic and relying on camel for sending/receiving requests/replies with the proper correlation.

Options

Name	Default Value	Description
spaceName	null	This is the JavaSpace name
verb	take	This is the verb for getting JavaSpace entries, it can be: take or read
transactional	false	if true, sending and receiving of entries is performed under a transaction
transactionalTimeout	Long.MAX_VALUE	the transaction timeout
concurrentConsumers	1	the number of concurrent consumer getting entries from the JavaSpace
templateId	null	if present, this option it's a Spring Bean id to be used as a template for reading/taking entries

Using camel-javaspace

To use this module you need to use the FUSE Mediation Router distribution. Or you could just add the following to your pom.xml, substituting the version number for the latest & greatest release.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-parent</artifactId>
  <version>1.4.0.0-fuse</version>
</dependency>
```

And ensure you are pointing at the maven repo

```
<repository>
     <id>open.iona.m2</id>
     <name>IONA Open Source Community Release Repository</name>
     <url>http://repo.open.iona.com/maven2</url>
     <snapshots>
       <enabled>false</enabled>
     </snapshots>
     <releases>
       <enabled>true</enabled>
     </releases>
   </repository>
```

Building From Source

The source for camel-javaspace is available here: https://projects.open.iona.com/projects/svn/iona/camel/trunk/components/camel-javaspace/

You'll need to register with http://open.iona.com to be able to access subversion.

The full FUSE distro is here: https://projects.open.iona.com/projects/svn/iona/camel/trunk/

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JBI COMPONENT

The **jbi:** component is provided by the ServiceMix Camel module and provides integration with a JBI Normalized Message Router such as provided by Apache ServiceMix

Following code

```
from("jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint")
```

automatically exposes new endpoint to the bus where service qname is {http://foo.bar.org}MyService and endpoint name is MyEndpoint (refer to URI format later).

All that has to be done to use this endpoint is to send messages from some endpoint already declared (for example with `jms:endpoint`) to this JBI camel endpoint (the same way as messages are sent to EIP endpoints or any other endpoint) and camel will pick it up the same way as it picks any other messages.

Sending works in the same way:

```
to("jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint")
```

Is used to send messages to JBI endpoint already deployed to the bus. It could be an endpoint exposed by `jms:endpoint`, `http:provider` or anything else.

URI format

```
jbi:service:serviceNamespace[sep]serviceName
jbi:endpoint:serviceNamespace[sep]serviceName[sep]endpointName
jbi:name:endpointName
```

The separator used will be:

- '/' if the namespace looks like 'http://'
- ':' if the namespace looks like 'urn:foo:bar'

For more details of valid JBI URIs see the ServiceMix URI Guide.

Using the **jbi:service:** or **jbi:endpoint:** URI forms will set the service QName on the JBI endpoint to the exact one you give. Otherwise the default Camel JBI Service QName will be used which is

```
{http://activemq.apache.org/camel/schema/jbi}endpoint
```

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
```

Creating a JBI Service Unit

If you have some Camel routes you want to deploy inside JBI as a Service Unit you can use the JBI Service Unit Archetype to create a new project.

If you have an existing maven project which you need to convert into a JBI Service Unit you may want to refer to the ServiceMix Maven JBI Plugins for further help. Basically you just need to make sure

- you have a spring XML file at **src/main/resources/camel-context.xml** which is used to boot up your routes inside the JBI Service Unit
- you change the pom'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/XMLSchema-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>
  <version>1.0-SNAPSHOT</version>

  <name>A Camel based JBI Service Unit</name>

  <url>http://www.myorganization.org</url>

  <properties>
    <camel-version>1.0.0</camel-version>
    <servicemix-version>3.2-incubating</servicemix-version>
  </properties>

  <dependencies>
    <dependency>
```

```
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-jbi</artifactId>
      <version>${camel-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>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- ServiceMix Camel module
- Using Camel with ServiceMix

JCR COMPONENT

The `jcr:` component allows you to add nodes to a JCR (JSR-170) compliant content repository (e.g. Apache Jackrabbit).

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 producer endpoint created by this component:

- a new node is created in the content repository
- all the message properties of the `in` message will be 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 `org.apache.camel.component.jcr.node_name` (you can refer to `JcrComponent.NODE_NAME` in your code), which is used to determine the node name.

Example

The snippet below will create a node named `node` under the `/home/test` node in the content repository. One additional attribute will be added to the node as well: `my.contents.property` will contain the body of the message being sent.

```
from("direct:a").setProperty(JcrComponent.NODE_NAME, constant("node"))
    .setProperty("my.contents.property", body()).to("jcr://user:pass@repository/
home/test");
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JDBC COMPONENT

The **jdbc:** component allows you to work with databases using JDBC queries and operations via SQL text as the message payload.
This component uses standard Java JDBC to work with the database, unlike the SQL Component component that uses spring-jdbc.

So far endpoints from this component could be used only as producers. It means that you cannot use them in `from()` statement.

URI format

```
jdbc:dataSourceName?options
```

This component only supports producer, meaning that you can not use routes with this component in the `from` type.

Options

Name	Default Value	Description
readSize	2000	The default maximum number of rows that can be read by a polling query

Result

The result is returned in the out body as a `ArrayList<HashMap<String, Object>>` list object with the result. The List contains the list of rows and the Map contains 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.

If the query is an update query the update count is returned in the header `jdbc.updateCount`

Samples

In the sample below we fetches the rows from the customer table.

First we register our datasource in the Camel 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, notice that 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");
}
```

And then we create the endpoint and sends the exchange containing the SQL query to execute in the in body. The result 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"));
```

Sample - Polling the database every minute

If we want to poll a database using this component we need to combine this with a polling scheduler such as the Timer or Quartz etc.
In this sample we retrieve data from the database every 60th seconds.

```
from("timer://foo?period=60000").setBody(constant("select * from
customer")).to("jdbc:testdb").to("activemq:queue:customers");
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JETTY COMPONENT

The **jetty:** component provides HTTP based endpoints for consuming HTTP requests that arrive at an http endpoint.

URI format

```
jetty:http:hostname[:port][/resourceUri][?options]
```

Options

Name	Description	Example	Required?	default value
sessionSupport	The option for enable the session manager in the server side of Jetty.	sessionSupport=true	No	false

Usage

You can only consume from endpoints generated by the Jetty component. Therefore it should only be used as input into your camel Routes. To issue HTTP requests against other HTTP endpoints you can use the HTTP Component

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

Example

```
from("timer://foo?fixedRate=true&delay=0&period=10000").to("jetty:http://www.google.com").set
"message.html").to("file:target/hello");
```

Poll the google homepage every 10 seconds and write the page to the file message.html

JING COMPONENT

The Jing component uses the Jing Library to perform XML validation of the message body using either

- RelaxNG XML Syntax
- RelaxNG Compact Syntax

Note that the MSV component can also support RelaxNG XML syntax.

URI format

```
rng:someLocalOrRemoteResource
rnc:someLocalOrRemoteResource
```

Where **rng** means use the RelaxNG XML Syntax whereas **rnc** means use RelaxNG Compact Syntax. The following examples show possible URI values

Example	Description
rng:foo/bar.rng	Will take the XML file **foo/bar.rng** on the classpath
rnc:http://foo.com/bar.rnc	Will use the RelaxNG Compact Syntax file from the URL http://foo.com/bar.rnc

Example

The following example 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 Compact Syntax schema (which is supplied on the classpath).

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <try>
      <to uri="rnc:org/apache/camel/component/validator/jing/schema.rnc"/>
      <to uri="mock:valid"/>

      <catch>
        <exception>org.apache.camel.ValidationException</exception>
        <to uri="mock:invalid"/>
      </catch>
    </try>
  </route>
</camelContext>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JMS COMPONENT

The JMS component allows messages to be sent to a JMS Queue or Topic; or messages to be consumed from a JMS Queue or Topic. The implementation of the JMS Component uses

Spring's JMS support for declarative transactions, using Spring's JmsTemplate for sending and a MessageListenerContainer for consuming.

URI format

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

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

```
jms:FOO.BAR
```

You can be completely specific if you wish via

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

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

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

Using Temporary Destinations

As of 1.4.0 of Camel you can use temporary queues using the following URL format

```
jms:temp:queue:foo
```

or temporary topics as

```
jms:temp:topic:bar
```

Where **foo** and **bar**, the text after the String **jms:temp:queue:** or **jms:temp:topic:**, are the names of the destinations. This enables multiple routes or processors or beans to refer to the same temporary destination. e.g. you can create 3 temporary destinations and use them in routes as inputs or outputs by referring to them by name.

Notes

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

When using message headers; the JMS specification states that header names must be valid Java identifiers. So by default camel will ignore any headers which do not match this rule. So try name your headers as if they are valid Java identifiers. One added bonus of this is that you can then use your headers inside a JMS Selector - which uses SQL92 syntax which mandates Java identifier syntax for headers.

From Camel 1.4 a simple strategy for mapping headers names is used by default. The strategy is to replace any dots in the headername with underscore, and vice-versa when the header name is restored from the JMS message that was sent over the wire. What does this means? No more loosing method names to invoke on a bean component, no more loosing the filename header for the File Component etc.

Current header name strategy used for accepting header names in Camel:

- replace all dots with underscores (e.g. `org.apache.camel.MethodName` => `org_apache_camel_MethodName`)
- test if the name is a valid java identifier using the JDK core classes
- if test success then the header is added and sent over the wire, if not its dropped (logged at DEBUG level)

Properties

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

Property	Default Value	Description
acceptMessagesWhileStopping	false	Should the consumer accept messages while it is stopping
acknowledgementModeName	"AUTO_ACKNOWLEDGE"	The JMS acknowledgement name which is one of: TRANSACTED, CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE
autoStartup	true	Should the consumer container auto-startup
cacheLevelName	"CACHE_CONNECTION" but when SPR-3890 is fixed it will be "CACHE_CONSUMER"	Sets the cache level name for the underlying JMS resources
clientId	null	Sets the JMS client ID to use. Note that this value if specified must be unique and can only be used by a single JMS connection instance. Its typically only required for durable topic subscriptions. You may prefer to use Virtual Topics instead

If you are using ActiveMQ

Note that the JMS component reuses Spring 2's JmsTemplate for sending messages. This is not ideal for use in a non-J2EE container and typically requires some caching JMS provider to avoid performance being lousy.

So if you intent to use Apache ActiveMQ as your Message Broker - which is a good choice as ActiveMQ rocks 🙂, then we recommend that you either

- use the ActiveMQ component which is already configured to use ActiveMQ efficiently
- use the PoolingConnectionFactory in ActiveMQ

For Consuming Messages cacheLevelName settings are vital!

If you are using Spring before 2.5.1 and Camel before 1.3.0 then you might want to set the **cacheLevelName** to be CACHE_CONSUMER for maximum performance.

Due to a bug in earlier Spring versions causing a lack of transactional integrity, previous versions of Camel and Camel versions from 1.3.0 onwwards when used with earlier Spring versions than 2.5.1 will default to use CACHE_CONNECTION. See the JIRAs CAMEL-163 and CAMEL-294.

Also if you are using XA or running in a J2EE container then you may want to set the **cacheLevelName** to be CACHE_NONE as we have seen using JBoss with TibCo EMS and JTA/XA you must disable caching.

concurrentConsumers	1	Specifies the default number of concurrent consumers
connectionFactory	null	The default JMS connection factory to use for the *listenerConnectionFactory* and *templateConnectionFactory* if neither are specified
deliveryPersistent	true	Is persistent delivery used by default?

disableReplyTo	false	Do you want to ignore the JMSReplyTo header and so treat messages as InOnly by default and not send a reply back?
durableSubscriptionName	null	The durable subscriber name for specifying durable topic subscriptions
exceptionListener	null	The JMS Exception Listener used to be notified of any underlying JMS exceptions
explicitQosEnabled	false	Set if the deliveryMode, priority or timeToLive should be used when sending messages
exposeListenerSession	true	Set if the listener session should be exposed when consuming messages
idleTaskExecutionLimit	1	Specify the limit for idle executions of a receive task, not having received any message within its execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in case of dynamic scheduling; see the "maxConcurrentConsumers" setting).
listenerConnectionFactory	null	The JMS connection factory used for consuming messages
maxConcurrentConsumers	1	Specifies the maximum number of concurrent consumers
maxMessagesPerTask	1	The number of messages per task
messageConverter	null	The Spring Message Converter

messageIdEnabled	true	When sending, should message IDs be added
messageTimestampEnabled	true	Should timestamps be enabled by default on sending messages
priority	-1	Values of > 1 specify the message priority when sending, if the explicitQosEnabled property is specified
selector	null	Sets the JMS Selector which is an SQL 92 predicate used to apply to messages to filter them at the message broker. You may have to encode special characters such as = as %3D
receiveTimeout	none	The timeout when receiving messages
recoveryInterval	none	The recovery interval
serverSessionFactory	null	The JMS ServerSessionFactory if you wish to use ServerSessionFactory for consumption
subscriptionDurable	false	Enabled by default if you specify a durableSubscriberName and a clientId
taskExecutor	null	Allows you to specify a custom task executor for consuming messages
templateConnectionFactory	null	The JMS connection factory used for sending messages
timeToLive	null	Is a time to live specified when sending messages
transacted	false	Is transacted mode used for sending/receiving messages?

transactionManager	null	The Spring transaction manager to use
transactionName	null	The name of the transaction to use
transactionTimeout	null	The timeout value of the transaction if using transacted mode
useVersion102	false	Should the old JMS API be used

Message format

The exchange that is sent over the JMS wire must conform to the JMS Message spec.

For the `exchange.in.header` the following rules apply for the **keys**:

- Keys stating with JMS or JMSX is reserved. All user keys will be dropped.
- `exchange.in.headers` keys must be literals and all be valid Java identifiers. (do not use dots in the key name)
- In Camel 1.4 onwards Camel will automatically replace all dots with underscore for key names. And vice-versa when Camel consumes JMS messages.

For the `exchange.in.header` the following rules apply for the **values**:

- The values must be primitives or their counter objects (such as Integer, Long, Character). String, CharSequence, Date, BigDecimal or BigInteger is all converted to their `toString()` representation. All other types is dropped.

Camel will log with category `org.apache.camel.component.jms.JmsBinding` at **DEBUG** level if it drops a given header value. Example:

```
2008-07-09 06:43:04,046 [main           ] DEBUG JmsBinding                     -
Ignoring non primitive header: order of class:
org.apache.camel.component.jms.issues.DummyOrder with value:
DummyOrder{orderId=333, itemId=4444, quantity=2}
```

Configuring different JMS providers

You can configure your JMS provider inside the Spring XML as follows...

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/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 via the id attribute**. The above example configures an '*activemq*' component. You could do the same to configure MQSeries, TibCo, BEA, Sonic etc.

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 as **activemq:[queue:|topic:]destinationName**. So you could use the same approach for working with 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.

Enabling Transacted Consumption

A common requirement is to consume from a queue in a transaction then process the message using the Camel route. To do this just ensure you set the following properties on the component/endpoint

- transacted = true
- transactionManager = a Transsaction Manager - typically the JmsTransactionManager

Using JNDI to find the ConnectionFactory

If you are using a J2EE container you might want to lookup in JNDI to find your ConnectionFactory rather than use the usual <bean> mechanism in spring. You can do this using Spring's factory bean or the new XML namespace. e.g.

```
<bean id="weblogic" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory" ref="myConnectionFactory"/>
</bean>

<jee:jndi-lookup id="myConnectionFactory" jndi-name="java:env/
ConnectionFactory"/>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JPA COMPONENT

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

Sending to the endpoint

Sending POJOs to the JPA endpoint inserts entities into the database. The body of the message is assumed to be an entity bean (i.e. a POJO with an @Entity annotation on it).

If the body does not contain an entity bean then 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 which will be invoked on your entity bean when the entity bean is consumed.

URI format

```
jpa:[entityClassName]
```

For sending to the endpoint, the **entityClassName** is optional. If specified it is used to help use the [Type Conversion] to ensure the body is of the correct type.

For consuming the **entityClassName** is mandatory.

Options

Name	Default Value	Description
entityType	entityClassName	Is the provided entityClassName from the URI
persistenceUnit	camel	the JPA persistence unit used by default
consumeDelete	true	Option for JpaConsumer only. Enables / disables whether or not the entity is deleted after it is consumed.

consumeLockEntity	true	Option for JpaConsumer only. Enables / disables whether or not to use exclusive locking of each entity while processing the results from the pooling.
flushOnSend	true	Option for JpaProducer only. Flushes the EntityManager after the entity beans has been persisted.
maximumResults	-1	Option for JpaConsumer only. Set the maximum number of results to retrieve on the Query.
consumer.delay	500	Option for JpaConsumer only. Delay in millis between each poll.
consumer.initialDelay	1000	Option for JpaConsumer only. Millis before polling starts.
consumer.userFixedDelay	false	Option for JpaConsumer only. **true** to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

LIST COMPONENT

The List component provdes 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

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

LOG COMPONENT

The **log:** component logs message exchanges to the underlying logging mechanism.
Camel uses commons-logging which allows you to configure logging via

- Log4j
- JDK 1.4 logging
- Avalon
- SimpleLog - a simple provider in commons-logging

Refer to the commons-logging user guide for a more complete overview of how to use and configure commons-logging.

URI format

```
log:loggingCategory[?level=loggingLevel][options]
```

Where **loggingCategory** is the name of the logging category to use and **loggingLevel** is the logging level such as DEBUG, INFO, WARN, ERROR - the default is INFO

Formatting

The log formats the execution of exchanges to log lines.
The log uses by default `LogFormatter` to format the log output.

TraceFormatter has the following options

Option	Default	Description
showExchangeId	false	To output the unique exchange id. Currently the breadcrumb is sufficient.
showProperties	false	Output the exchange properties
showHeaders	false	Output the in message headers
showBodyType	true	Output the in body Java type
showBody	true	Output the in body
showOut	false	If the exchange has an out message then its also shown
showAll	false	quick option for turning all options on

multiline	false	if **enabled** then each information is logged on a newline

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- How do I use log4j
- How do I use Java 1.4 logging

MAIL COMPONENT

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

URI format

```
smtp://[user-info@]host:port[?password=somepwd]
pop3://[user-info@]host:port[?password=somepwd]
imap://[user-info@]host:port[?password=somepwd]
```

which supports either POP, IMAP or SMTP underlying protocols.

It is possible to omit the user-info and specify the username as a URI parameter instead

```
smtp://host:port?password=somepwd&username=someuser
```

Such as:

```
smtp://mycompany.mailserver:30?password=tiger&username=scott
```

SSL support

Camel have support for secure mail protocols. Just add a s to the scheme such as:

```
smtps://[user-info@]host:port[?password=somepwd]
pop3s://[user-info@]host:port[?password=somepwd]
imaps://[user-info@]host:port[?password=somepwd]
```

Mail Message Content

Camel will use the Exchange Message IN body as the MimeMessage text content. The body is converted to String.class.

Camel copies all the Exchange Message IN headers to the MimeMessage headers.

Default behavior changed

As of Camel 1.4 the default consumer delay is now 60 seconds. Camel will therefore only poll the mailbox once a minute to not exhaust the mail server. The default value in Camel 1.3 is 500 millis.

SSL Information

Its the underlying mail framework that is handling the SSL support. Camel uses SUN JavaMail. However SUN JavaMail only trusts certificates issued by well known Certificate Authorities. So if you have issued your own certificate you have to import it into the local Java keystore file (see SSLNOTES.txt in JavaMail for details).

If you are using your own signed certificates sometimes it can be cumbersome to install your certificate in the local keystore. Camel provides a test option **dummyTrustManager** that when enabled will accept any given certificate. **Notice:** this is strongly discouraged not using in production environments.

The subject of the MimeMessage can be configured using a header property on the in message. The code below demonstrates this:

```
from("direct:a").setHeader("subject", subject).to("smtp://james2@localhost");
```

Default Ports

As of Camel 1.4 support for default port number has been added. If the port number is omitted Camel will determine based on the protocol the port number to use.

Protocol	Default Port Number
SMPT	25
SMPTS	465
POP3	110
POP3S	995
IMAP	143
IMAPS	993

Options

Property	Default	Description
host	Ê	The host name or IP address to connect to
port	See Default Ports	The TCP port number to connect on
user-info	Ê	The user name on the email server
username	Ê	The user name on the email server configured as a URI parameter
password	null	The users password to use, can be omitted if the mail server does not require a password
ignoreUriScheme	false	If enabled Camel will use the scheme to determine the transport protocol to use (pop, imap, smtp etc.)
defaultEncoding	null	The default encoding to use for MineMessages
folderName	INBOX	The folder to poll
destination	user-info@host	@deprecated use To option. The TO recipients (the receivers of the mail)
to	user-info@host	The TO recipients (the receivers of the mail). This option is introduced in Camel 1.4.
CC	null	The CC recipients (the receivers of the mail). This option is introduced in Camel 1.4.
BCC	null	The BCC recipients (the receivers of the mail). This option is introduced in Camel 1.4.
from	camel@localhost	The FROM email address
deleteProcessedMessages	true	Deletes the messages after they have been processed. This is done by setting the DELETED flag on the mail message. If **false** then the flag SEEN is set instead.

processOnlyUnseenMessages	false	As of Camel 1.4 its possible to configure MailConsumer to only process unseen messages (eg new messages) or all. Note Camel will always skip deleted messages. Setting this option to **true** will filter to only unseen messages.
fetchSize	-1	As of Camel 1.4 a maximum number of messages to consume during a polling can be set. This can be used to not exhaust a mail server if a mailbox folder contains a lot of messages. Default value of -1 means no fetch size and all messages will be consumed. Setting the value to 0 is a special corner case where Camel will not consume any messages at all.
debugMode	false	As of Camel 1.4 its possible to enable the debug mode on the underlying mail framework. SUN Mail framework will default output to System.out.
connectionTimeout	30000	As of Camel 1.4 the connection timeout can be configured in millis. Default is 30 seconds.
dummyTrustManager	false	As of Camel 1.4 testing SSL connections can be easier if enabling a dummy TrustManager that trust any given certificate. Notice this is only to be used for testing, as it does not provide any security at all.
consumer.initialDelay	1000	Millis before the polling starts
consumer.delay	60000	As of Camel 1.4 the default consumer delay is now 60 seconds. Camel will therefore only poll the mailbox once a minute to not exhaust the mail server. The default value in Camel 1.3 is 500 millis.

consumer.useFixedDelay	false	**true** to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.

Samples

We start with a simple route that sends the messages received from a JMS queue as emails. The email account with be the admin account on mymailserver.com.

```
from("jms://queue:subscription").to("smtp://admin@mymailserver.com?password=secret");
```

In the next sample we will poll a mailbox for new emails once every minute. Notice that we use the special consumer parameter for setting the poll interval **consumer.delay** as 60000 millis = 60 seconds.

```
from("imap://admin@mymailserver.com?password=secret&processOnlyUnseenMessages=true&consumer.d
```

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@somewhere.org";

from("direct:a").to("smtp://you@mymailserver.com?password=secret&From=you@apache.org"
+ recipients);
```

Attachment Sample

Attachments is a new feature in Camel 1.4 that of course is also supported by the mail component. 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. Google mail requires to use SSL and have it configured for other clients to access your mailbox. This is done by logging into your google mail and change your settings to allow IMAP access. Google have extensive documentation how to do this.

```
from("imaps://imap.gmail.com?username=YOUR_USERNAME@gmail.com&password=YOUR_PASSWORD&deletePr
```

The route above will poll the google mail inbox for new mails once every minute and log it 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=[Camel mail component is now much better],
sentDate=[2008-05-08 06:31:44], receivedDate=[2008-05-08 06:34:39]
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=[Camel mail component is now much better],
sentDate=[2008-05-08 06:31:44], receivedDate=[2008-05-08 06:34:39]]
```

SSL Sample with dummyTrustManager

In the next sample we want to sent mails from Camel using our own mail server using secure connections. As our own mail server is using our own signed certificate we need either to

1. install our certificate in the local keystore Camel uses
2. use the dummyTrustManager option for testing purpose to see if the secure communication works

In the sample we use the dummyTrustManager option:

```
from("seda:mailsToSend").to("imaps://ourmailsserver.com?username=camelmail&password=secret&du
```

See Also

- Configuring Camel

- Component
- Endpoint
- Getting Started

MINA COMPONENT

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

URI format

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

From Camel 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.

For UDP/Multicast if no codec is specified the default uses a basic ByteBuffer based codec.

Multicast also has a shorthand notation **mcast**.

The VM protocol is used as a direct forwarding mechanism in the same JVM. See the MINA VM-Pipe API documentation for details.

A MinaProducer has a default timeout value of 30 seconds, while it waits for a response from the remote server.

In normal usage Camel-mina only supports marshalling the body content - message headers and exchange properties will not be sent.
However the option **transferExchange** does allow to transfer the exchange itself over the wire. See options below.

Options

Name	Default Value	Description
codec	null	As of 1.3 or later you can refer to a named ProtocolCodecFactory instance in your Registry such as your Spring ApplicationContext which is then used for the marshalling
textline	null	Only used for TCP. If no codec is specified then 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.

sync	false	As of 1.3 or later 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).
lazySessionCreation	false	As of 1.3 or later session can be lazy created to avoid exceptions if the remote server is not up and running when the Camel producer is started.
timeout	30000	As of 1.3 or later you can configure the timeout while waiting for a response from a remote server. The timeout unit is in millis, so 60000 is 60 seconds. The timeout is only used for MinaProducer.
encoding	JVM Default	As of 1.3 or later you can configure the encoding (is a charset name) to use for the TCP textline codec and the UDP protocol. If not provided Camel will use the JVM default Charset.
transferExchange	false	Only used for TCP. As of 1.3 or later you can transfer the exchange over the wire instead of just the body. The following fields is transfered: in body, out body, in headers, out headers, exchange properties, exchange exception.
minaLogger	false	As of 1.3 or later you can enable Apache MINA logging filter. Apache MINA uses slf4j logging at INFO level to log all input and output.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

MOCK COMPONENT

Testing of distributed and asynchronous processing is notoriously difficult. The Mock, Test and DataSet endpoints work great with the Spring Testing framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's large range of Components together with the powerful Mock and Test testing endpoints.
The Mock component provides a powerful declarative testing mechanism which is similar to jMock in that it allows declarative expectations to be created on any Mock endpoint before a

test begins. Then the test is ran 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
- that the correct payloads are received, in the right order
- that messages arrive on an endpoint in order, using some Expression to create an order testing function
- that 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 which is-a Mock endpoint but which also uses a second endpoint to provide the list of expected message bodies and automatically sets up the Mock endpoint assertions. i.e. its a Mock endpoint which automatically sets up its assertions from some sample messages in a File or database for example.

URI format

```
mock:someName
```

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

Simple Example

Here's a simple example of MockEndpoint in use. First the endpoint is resolved on the context. Then we set an expectation, then after the test has run we assert our expectations are 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 to test that the expectations were met after running a test.

Setting expectations

You can see from the javadoc of MockEndpoint the various helper methods you can use to set expectations. The main methods available are as follows

Method	**Description**

expectedMessageCount(int)	to define the expected message count on the endpoint
expectedMinimumMessageCount(int)	to define the minimum number of expected messages on the endpoint
expectedBodiesReceived(...)	to define the expected bodies that should be received (in order)
expectsAscending(Expression)	to add an expectation that messages are received in order using the given Expression to compare messages
expectsDescending(Expression)	to add an expectation that messages are received in order using the given Expression to compare messages
expectsNoDuplicates(Expression)	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",
"thirdMessageBody");
```

Adding expectations to specific messages

In addition you can use the message(int messageIndex) 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 this code

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

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

A Spring Example

First here's the spring.xml file

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="file:src/test/data?noop=true"/>
    <filter>
      <xpath>/person/city = 'London'</xpath>
      <to uri="mock:matched"/>
    </filter>
  </route>
</camelContext>

<bean id="myBean" class="org.apache.camel.spring.mock.MyAssertions"
scope="singleton"/>
```

As you can see it defines a simple routing rule which consumes messages from the local src/test/data directory. The **noop** flag just means not to delete or move the file after its been processed.

Also note we instantiate a bean called **myBean**, here is the source of the MyAssertions bean.

```
public class MyAssertions implements InitializingBean {
    @EndpointInject(uri = "mock:matched")
    private MockEndpoint matched;

    @EndpointInject(uri = "mock:notMatched")
    private MockEndpoint notMatched;

    public void afterPropertiesSet() throws Exception {
        // lets add some expectations
        matched.expectedMessageCount(1);
        notMatched.expectedMessageCount(0);
    }

    public void assertEndpointsValid() throws Exception {
        // now lets perform some assertions that the test worked as we expect
        Assert.assertNotNull("Should have a matched endpoint", matched);
        Assert.assertNotNull("Should have a notMatched endpoint", notMatched);
        MockEndpoint.assertIsSatisfied(matched, notMatched);
    }
}
```

The bean is injected with a bunch of Mock endpoints using the @EndpointInject annotation, it then sets a bunch of expectations on startup (using Spring's InitializingBean interface and afterPropertiesSet() method) before the CamelContext starts up.

Then in our test case (which could be JUnit or TesNG) we lookup **myBean** in Spring (or have it injected into our test) and then invoke the **assertEndpointsValid()** method on it to verify that the mock endpoints have their assertions met. You could then inspect the message exchanges that were delivered to any of the endpoints using the getReceivedExchanges() method on the Mock endpoint and perform further assertions or debug logging.

Here is the actual JUnit test case we use.

See Also

- Configuring Camel
- Component

- Endpoint
- Getting Started
- Spring Testing

MSV COMPONENT

The MSV component performs XML validation of the message body using the MSV Library using any of the XML schema languages supported such as XML Schema or RelaxNG XML Syntax.

Note that the Jing component also supports RelaxNG Compact Syntax

URI format

```
msv: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. For example

- msv:org/foo/bar.rng
- msv:file:../foo/bar.rng
- msv:http://acme.com/cheese.rng

Example

The following example 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 (which is supplied on the classpath).

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <try>
      <to uri="msv:org/apache/camel/component/validator/msv/schema.rng"/>
      <to uri="mock:valid"/>

      <catch>
        <exception>org.apache.camel.ValidationException</exception>
        <to uri="mock:invalid"/>
      </catch>
    </try>
  </route>
</camelContext>
```

See Also

- Configuring Camel
- Component
- Endpoint

- Getting Started

POJO COMPONENT

The **pojo:** component is now just an alias for the Bean component.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

QUARTZ COMPONENT

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

URI format

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

You can configure the Trigger and JobDetail using the parameters

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

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

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

Using Cron Triggers

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

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

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

which is equivalent to using the cron expression

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

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

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

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Timer

QUEUE COMPONENT

The **queue:** component provides asynchronous SEDA behaviour so that messages are exchanged on a BlockingQueue 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 component.

Note also that this component has nothing to do with JMS, if you want a distributed SEA then try using either JMS or ActiveMQ or even MINA

URI format

```
queue:someName
```

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

See Also

- Configuring Camel
- Component
- Endpoint

Deprecated

To avoid confusion with JMS queues, this component is now deprecated in 1.1 onwards. Please use the SEDA component instead

- Getting Started

REF COMPONENT

The **ref:** component is used for lookup of existing endpoints bound in the Registry.

URI format

```
ref:someName
```

Where **someName** can be any string which is used to lookup the endpoint in the Registry.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

RMI COMPONENT

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

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

URI format

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

For example:

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

Using

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

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

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

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

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

Options

Name	Default Value	Description
method	null	As of 1.3 or later you can set the name of the method to invoke

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

SEDA COMPONENT

The **seda:** component provides asynchronous SEDA behaviour so that messages are exchanged on a BlockingQueue and consumers are invoked in a seperate thread to the producer. 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 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 endpoint with a thread pool which can process messages both synchronously and asynchronously. For example, from(direct:stageName").thread(5).process(..).

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 component.

This component does not implement any kind of persistence or recovery if the VM termininates while messages are yet to be processed. If you need persistence, reliability or distributed SEDA then try using either JMS or ActiveMQ

URI format

```
seda:someName
```

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

URI Options

Name	Description
size	The maximum size of the SEDA queue

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

SPRING INTEGRATION COMPONENT

The **spring-integration:** component provides a bridge for Camel components to talk to spring integration endpoints.

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.

Options

Name	Description	Example	Required	de val

inputChannel	The spring integration input channel name this endpoint wants to consume from that is defined in the spring context	inputChannel=requestChannel	No	Ê
outputChannel	The spring integration output channel name to send message to the spring integration context	outputChannel=replyChannel	No	Ê
inOut	The exchange pattern that spring integration endpoint should use	inOut=true	No	inC the inte cor and out for spr inte pro
consumer.delay	Delay in millis between each poll	consumer.delay=60000	No	500
consumer.initialDelay	Millis before polling starts	consumer.initialDelay=10000	No	100
consumer.userFixedDelay	true to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.	consumer.userFixedDelay=false	No	fals

Usage

Spring Integration component is a bridge which connects Spring Integration endpoints through the Spring integration's input and output channels with the Camel endpoints. In this way, we can send out the Camel message to Spring Integration endpoints or receive the message from Spring Integration endpoint in Camel routing context.

Examples

Using the Spring Integration Endpoint

You could setup the Spring Integration Endpoint by using the URI

```
<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-2.5.xsd
            http://www.springframework.org/schema/integration
            http://www.springframework.org/schema/integration/
spring-integration-1.0.xsd
            http://activemq.apache.org/camel/schema/spring
            http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">

    <message-bus auto-create-channels="true"/>

    <handler-endpoint input-channel="inputChannel"
              output-channel="outputChannel"
              ref="helloService"
              method="sayHello"/>

    <beans:bean id="helloService"
class="org.apache.camel.component.spring.integration.HelloWorldService"/>
    <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/
spring">
      <route>
        <from uri="direct:start"/>
        <!-- Using the &amp; as the separator of & -->
        <to
uri="spring-integration:inputChannel?inOut=true&amp;inputChannel=outputChannel"/>
      </route>
    </camelContext>

<message-bus auto-create-channels="true"/>

   <channel id="requestChannel"/>
   <channel id="responseChannel"/>

   <beans:bean id="myProcessor"
class="org.apache.camel.component.spring.integration.MyProcessor"/>

   <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/
spring">
     <route>
       <!-- Using the &amp; as the separator of & -->
       <from
uri="spring-integration://requestChannel?outputChannel=responseChannel&amp;inOut=true&amp;con
       <process ref="myProcessor"/>
     </route>
   </camelContext>
```

or by the 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-2.5.xsd
            http://www.springframework.org/schema/integration
            http://www.springframework.org/schema/integration/
spring-integration-1.0.xsd
            http://activemq.apache.org/camel/schema/spring
            http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">
    <message-bus auto-create-channels="true"/>

    <channel id="outputChannel"/>

    <camelContext id="camel" xmlns="http://activemq.apache.org/camel/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 Integartion also provides the Spring Integration's Source and Target adapters which could route the message from the Spring Integration channel to a camel context endpoint or from a camel context endpoint to a Spring Integration Channel.

Here is the name spaces header

```
<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://activemq.apache.org/camel/schema/spring/
integration"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/integration
       http://www.springframework.org/schema/integration/
spring-integration-1.0.xsd
       http://activemq.apache.org/camel/schema/spring/integration
       http://activemq.apache.org/camel/schema/spring/integration/
camel-spring-integration.xsd
       http://activemq.apache.org/camel/schema/spring
```

```
        http://activemq.apache.org/camel/schema/spring/camel-spring.xsd
    ">
```

Now you could bind your source or target to camel context endpoint

```
<!-- Create the camel context here -->
<camelContext id="camelTargetContext" xmlns="http://activemq.apache.org/camel/
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"
requestChannel="channelA" expectReply="false">
    <camel-si:camelContextRef>camelTargetContext</camel-si:camelContextRef>
</camel-si:camelTarget>

<camel-si:camelTarget id="camelTargetB" camelEndpointUri="direct:EndpointC"
requestChannel="channelB" replyChannel="channelC" expectReply="true">
    <camel-si:camelContextRef>camelTargetContext</camel-si:camelContextRef>
</camel-si:camelTarget>

<beans:bean id="myProcessor"
class="org.apache.camel.component.spring.integration.MyProcessor"/>

<camelContext id="camelSourceContext" xmlns="http://activemq.apache.org/camel/
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 requestChannel 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 requestChannel channelB
 then it will pull the response from channelC and put the response message back
to direct:EndpointC -->

<camel-si:camelSource id="camelSourceB" camelEndpointUri="direct:EndpointC"
requestChannel="channelB" replyChannel="channelC" expectReply="true">
    <camel-si:camelContextRef>camelSourceContext</camel-si:camelContextRef>
</camel-si:camelSource>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

STREAM COMPONENT

The **stream:** component provides access to the System.in, System.out and System.err streams together with allowing streaming of output to a file.
Notice that some of the stream types has been @deprecated (file and url). These types are supported by their respective components.

URI format

```
stream:in
stream:out
stream:err
stream:file?file=/foo/bar.txt  (@deprecated)
stream:url (@deprecated)
stream:header
```

If the **stream:header** option is specified then the **stream** header is used to find the stream to write to. This option is only available for StreamProducer.

Options

Name	Default Value	Description
file	null	When using the **stream:file** notation this specifies the file name to stream to/from. @deprecated.
delay	0	Initial delay in millis before consuming or producing the stream.

encoding	JVM Default	As of 1.4 or later you can configure the encoding (is a charset name) to use text based streams (eg. message body is a String object). If not provided Camel will use the JVM default Charset.

Message content

The **stream:** component supports either String or byte[] for writing to streams. Just add to the `message.in.body` either a Stirng or byte[] content.
The special **stream:header** URI is used for custom output streams. Just add a `java.io.OutputStream` to `message.in.header` in the key `header`.
See samples for an example.

Samples

In this sample we output to System.out the content from the message when its put on the direct:in queue.

```
public void testStringContent() throws Exception {
    template.sendBody("direct:in", "Hello Text World\n");
}

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");
        }
    };
}
```

This 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();

public void testStringContent() {
    template.sendBody("direct:in", "Hello");
    // StreamProducer appends \n in text mode
    assertEquals("Hello\n", sb.toString());
}

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 {
        char c = (char)b;
        sb.append((char)b);
    }
}
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

STRING TEMPLATE

The **string-template:** component allows you to process a message using a String Template. This can be ideal when using Templating to generate responses for requests.

URI format

```
string-template:templateName
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template.

Options

Option	Default	Description
contentCache	false	New option in Camel 1.4. Cache for the resource content when its loaded.

Headers

Camel will store a reference to the resource in the message header in the key `org.apache.camel.stringtemplate.resource`. The Resource is an `org.springframework.core.io.Resource` object.

Hot reloading

The stringtemplate resource is by default hot reloadable for both file and classpath resources (expanded jar). Setting the contentCache=true then Camel will only load the resource once, and thus hot reloading is not possible. This scenario can be used in production usage when the resource never changes.

StringTemplate Attributes

Camel will provide exchange information as attributes (just a 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 something like

```
from("activemq:My.Queue").
  to("string-template:com/acme/MyResponse.tm");
```

To use a string template to formulate a response for a message

The Email Sample

In this sample we want to use StringTemplate as templating for 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:

```
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;
}

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 Action.\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("string-template:org/apache/camel/component/
stringtemplate/letter.tm").to("mock:result");
        }
    };
}
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

TEST COMPONENT

Testing of distributed and asynchronous processing is notoriously difficult. The Mock, Test and DataSet endpoints work great with the Spring Testing framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's large range of Components together with the powerful Mock and Test testing endpoints.
The Test component extends the Mock component to support pulling messages from another endpoint on startup to set the expected message bodies on the underlying Mock endpoint.

i.e. 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 setup a properly configured Mock 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 where 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 then your test case will perform the necessary assertions.

Here is a real example test case using Mock and Spring along with its Spring XML.

To see how you can set other expectations on the test endpoint, see the Mock component.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Spring Testing

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

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

Using

To setup a route that generates an event every 60 seconds:

```
from("timer://foo?fixedRate=true&delay=0&period=60000").to("bean:myBean?methodName=someMethod
```

The above route will generate an event then invoke the someMethodName on the bean called myBean in the Registry such as JNDI or Spring.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

- Quartz

VALIDATION COMPONENT

The Validation component performs XML validation of the message body using the JAXP Validation API using any of the supported XML schema languages, which defaults to XML Schema

Note that the Jing component also supports the following schema languages which are useful

- RelaxNG Compact Syntax
- RelaxNG XML Syntax

The MSV component also supports RelaxNG XML Syntax.

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

Example

The following example 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://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <try>
      <to uri="validator:org/apache/camel/component/validator/schema.xsd"/>
      <to uri="mock:valid"/>

      <catch>
        <exception>org.apache.camel.ValidationException</exception>
        <to uri="mock:invalid"/>
      </catch>
    </try>
  </route>
</camelContext>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

VELOCITY

The **velocity:** component allows you to process a message using an Apache Velocity template. This can be ideal when using Templating to generate responses for requests.

URI format

```
velocity:templateName
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (eg: file://folder/myfile.vm).

Options

Option	Default	Description
loaderCache	true	Velocity based file loader cache
contentCache	false	New option in Camel 1.4. Cache for the resource content when its loaded.

Headers

Camel will store a reference to the resource in the message header in the key `org.apache.camel.velocity.resource`. The Resource is an `org.springframework.core.io.Resource` object.

In Camel 1.4 headers set during the velocity evaluation is returned to the message and added as headers. Then its kinda possible to return values from Velocity to the Message.

An example: Set the header value of fruit in the Velocity template .tm:

```
$in.setHeader('fruit', 'Apple')
```

The header 'fruit' is now accessible from the `message.out.headers`.

Velocity Context

Camel 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
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). Setting the contentCache=true then Camel will only load the resource once, and thus hot reloading is not possible. This scenario can be used in production usage when the resource never changes.

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 for 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

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

And to use content cache, eg. for production usage 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");
```

The Email Sample

In this sample we want to use Velocity as 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;
}

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 Action.\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/
letter.vm").to("mock:result");
        }
    };
}
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

VM COMPONENT

The **vm:** component provides asynchronous SEDA behaviour so that messages are exchanged on a BlockingQueue and consumers are invoked in a seperate thread pool to the producer.

This component differs from the Queue component in that VM supports communcation 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

URI format

```
vm:someName
```

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)

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

XMPP COMPONENT

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

URI format

```
xmpp://[login@]hostname[:port][/room][?Options]
```

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

Example:

```
xmpp://superman@jabber.org/room33
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

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

```
from("activemq:My.Queue").
  to("xquery:com/acme/mytransform.xquery");
```

To use a xquery template to forumulate a response for 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

```
from("activemq:My.Queue").
  to("xquery:com/acme/mytransform.xquery").
  to("activemq:Another.Queue");
```

Options

Name	Default Value	Description

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

XSLT

The **xslt:** component allows you to process a message using an XSLT template. This can be ideal when using Templating to generate respopnses for requests.

URI format

```
xslt:templateName
```

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

Here are some example URIs

URI	Description
`xslt:com/acme/mytransform.xsl`	refers to the file com/acme/mytransform.xsl on the classpath
`xslt:file:///foo/bar.xsl`	refers to the file /foo/bar.xsl
`xslt:http://acme.com/cheese/foo.xsl`	refers to the remote http resource

Using XSLT endpoints

For example you could use something like

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl");
```

To use a xslt template to forumulate a response for 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

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl").
  to("activemq:Another.Queue");
```

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 along with its Spring XML if you want a concrete example.

Options

Name	Default Value	Description
converter	null	Option to override default XmlConverter. Will lookup for the converter in the Registry. The provided converted must be of type org.apache.camel.converter.jaxp.XmlConverter.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started