

-

USER GUIDE

Version 2.20.0

Copyright 2007-2016, Apache Software Foundation

Table of Contents

Table of Contents

- Chapter 1 - **Introduction**
- Chapter 2 - **Quickstart**
- Chapter 3 - **Getting Started**
- Chapter 4 - **Architecture**
- Chapter 5 - **Enterprise Integration Patterns**
- Chapter 6 - **Cook Book**
- Chapter 7 - **Tutorials**
- Chapter 8 - **Language Appendix**
- Chapter 9 - **DataFormat Appendix**
- Chapter 10 - **Pattern Appendix**
- Chapter 11 - **Component Appendix**
- Index**

CHAPTER 1**Introduction**

Apache Camel™ is a versatile open-source integration framework based on known Enterprise Integration Patterns. Camel empowers you to define routing and mediation rules in a variety of domain-specific languages, including a Java-based Fluent API, Spring or Blueprint XML Configuration files, and a Scala DSL. This means you get smart completion of routing rules in your IDE, whether in a Java, Scala or XML editor.

Apache Camel uses URIs to work directly with any kind of Transport or messaging model such as HTTP, ActiveMQ, JMS, JBI, SCA, MINA or CXF, as well as pluggable Components and Data Format options. Apache Camel is a small library with minimal dependencies for easy embedding in any Java application. Apache Camel lets you work with the same API regardless which kind of Transport is used - so learn the API once and you can interact with all the Components provided out-of-box.

Apache Camel provides support for Bean Binding and seamless integration with popular frameworks such as CDI, Spring, Blueprint and Guice. Camel also has extensive support for unit testing your routes.

The following projects can leverage Apache Camel as a routing and mediation engine:

- Apache ServiceMix - a popular distributed open source ESB and JBI container
- Apache ActiveMQ - a mature, widely used open source message broker
- Apache CXF - a smart web services suite (JAX-WS and JAX-RS)
- Apache Karaf - a small OSGi based runtime in which applications can be deployed
- Apache MINA - a high-performance NIO-driven networking framework

So don't get the hump - try Camel today! 😊

Too many buzzwords - what exactly is Camel?

Okay, so the description above is technology focused.

There's a great discussion about Camel at Stack Overflow. We suggest you view the post, read the comments, and browse the suggested links for more details.

CHAPTER 2**Quickstart**

To start using Apache Camel quickly, you can read through some simple examples in this chapter. For readers who would like a more thorough introduction, please skip ahead to Chapter 3.

WALK THROUGH AN EXAMPLE CODE

This mini-guide takes you through the source code of a simple example.

Camel can be configured either by using Spring or directly in Java - which this example does.

This example is available in the `examples\camel-example-jms-file` directory of the Camel distribution.

We start with creating a CamelContext - which is a container for Components, Routes etc:

```
{snippet:id=e1|lang=java|url=camel/trunk/examples/camel-example-jms-
file/src/main/java/org/apache/camel/example/jmstofile/CamelJmsToFileExample.java}There is more than one way of adding a
Component to the CamelContext. You can add components implicitly - when we set up the routing - as we do here for the
FileComponent:{snippet:id=e3|lang=java|url=camel/trunk/examples/camel-example-jms-
file/src/main/java/org/apache/camel/example/jmstofile/CamelJmsToFileExample.java}or explicitly - as we do here when we add
the JMS Component:{snippet:id=e2|lang=java|url=camel/trunk/examples/camel-example-jms-
file/src/main/java/org/apache/camel/example/jmstofile/CamelJmsToFileExample.java}The above works with any JMS provider. If
we know we are using ActiveMQ we can use an even simpler form using the activeMQComponent() method while specifying the
brokerURL used to connect to ActiveMQ
```

In normal use, an external system would be firing messages or events directly into Camel through one of its Components but we are going to use the ProducerTemplate which is a really easy way for testing your configuration:

```
{snippet:id=e4|lang=java|url=camel/trunk/examples/camel-example-jms-
file/src/main/java/org/apache/camel/example/jmstofile/CamelJmsToFileExample.java}Next you must start the camel context. If
you are using Spring to configure the camel context this is automatically done for you; though if you are using a pure Java
approach then you just need to call the start() method
```

`camelContext.start();`

This will start all of the configured routing rules.

So after starting the CamelContext, we can fire some objects into camel:

```
{snippet:id=e5|lang=java|url=camel/trunk/examples/camel-example-jms-
file/src/main/java/org/apache/camel/example/jmstofile/CamelJmsToFileExample.java}
```

WHAT HAPPENS?

From the ProducerTemplate - we send objects (in this case text) into the CamelContext to the Component `test-jms:queue:test.queue`. These text objects will be converted automatically into JMS Messages and posted to a JMS Queue named `test.queue`. When we set up the Route, we configured the FileComponent to listen off the `test.queue`.

The File FileComponent will take messages off the Queue, and save them to a directory named `test`. Every message will be saved in a file that corresponds to its destination and message id.

Finally, we configured our own listener in the Route - to take notifications from the FileComponent and print them out as text.

That's it!

If you have the time then use 5 more minutes to Walk through another example that demonstrates the Spring DSL (XML based) routing.

WALK THROUGH ANOTHER EXAMPLE**Introduction**

Continuing the walk from our first example, we take a closer look at the routing and explain a few pointers - so you won't walk into a bear trap, but can enjoy an after-hours walk to the local pub for a large beer 😊

First we take a moment to look at the Enterprise Integration Patterns - the base pattern catalog for integration scenarios. In particular we focus on Pipes and filters - a central pattern. This is used to route messages through a sequence of processing steps, each performing a specific function - much like the Java Servlet Filters.

Pipes and filters

In this sample we want to process a message in a sequence of steps where each steps can perform their specific function. In our example we have a JMS queue for receiving new orders. When an order is received we need to process it in several steps:

- validate
- register
- send confirm email

This can be created in a route like this:

```
<route>
  <from uri="jms:queue:order"/>
  <pipeline>
    <bean ref="validateOrder"/>
    <bean ref="registerOrder"/>
    <bean ref="sendConfirmEmail"/>
  </pipeline>
</route>
```

Pipeline is default

In the route above we specify `pipeline` but it can be omitted as its default, so you can write the route as:

```
<route>
  <from uri="jms:queue:order"/>
  <bean ref="validateOrder"/>
  <bean ref="registerOrder"/>
  <bean ref="sendConfirmEmail"/>
</route>
```

This is commonly used not to state the pipeline.

An example where the pipeline needs to be used, is when using a multicast and "one" of the endpoints to send to (as a logical group) is a pipeline of other endpoints. For example.

```
<route>
  <from uri="jms:queue:order"/>
  <multicast>
    <to uri="log:org.company.log.Category"/>
    <pipeline>
      <bean ref="validateOrder"/>
      <bean ref="registerOrder"/>
      <bean ref="sendConfirmEmail"/>
    </pipeline>
  </multicast>
</route>
```

The above sends the order (from `jms:queue:order`) to two locations at the same time, our log component, and to the "pipeline" of beans which goes one to the other. If you consider the opposite, sans the `<pipeline>`

```
<route>
  <from uri="jms:queue:order"/>
  <multicast>
    <to uri="log:org.company.log.Category"/>
    <bean ref="validateOrder"/>
    <bean ref="registerOrder"/>
    <bean ref="sendConfirmEmail"/>
  </multicast>
</route>
```

you would see that multicast would not "flow" the message from one bean to the next, but rather send the order to all 4 endpoints (1x log, 3x bean) in parallel, which is not (for this example) what we want. We need the message to flow to the validateOrder, then to the registerOrder, then the sendConfirmEmail so adding the pipeline, provides this facility.

Where as the `bean ref` is a reference for a spring bean id, so we define our beans using regular Spring XML as:

```
<bean id="validateOrder" class="com.mycompany.MyOrderValidator"/>
```

Our validator bean is a plain POJO that has no dependencies to Camel what so ever. So you can implement this POJO as you like. Camel uses rather intelligent Bean Binding to invoke your POJO with the payload of the received message. In this example we will **not** dig into this how this happens. You should return to this topic later when you got some hands on experience with Camel how it can easily bind routing using your existing POJO beans.

So what happens in the route above. Well when an order is received from the JMS queue the message is routed like Pipes and filters:

1. payload from the JMS is sent as input to the validateOrder bean
2. the output from validateOrder bean is sent as input to the registerOrder bean
3. the output from registerOrder bean is sent as input to the sendConfirmEmail bean

Using Camel Components

In the route lets imagine that the registration of the order has to be done by sending data to a TCP socket that could be a big mainframe. As Camel has many Components we will use the camel-mina component that supports TCP connectivity. So we change the route to:

```
<route>
  <from uri="jms:queue:order"/>
  <bean ref="validateOrder"/>
  <to uri="mina:tcp://mainframeip:4444?textline=true"/>
  <bean ref="sendConfirmEmail"/>
</route>
```

What we now have in the route is a `to` type that can be used as a direct replacement for the bean type. The steps is now:

1. payload from the JMS is sent as input to the validateOrder bean
2. the output from validateOrder bean is sent as text to the mainframe using TCP
3. the output from mainframe is sent back as input to the sendConfirmEmail bean

What to notice here is that the `to` is not the end of the route (the word 😊) in this example it's used in the middle of the Pipes and filters. In fact we can change the bean types to `to` as well:

```
<route>
  <from uri="jms:queue:order"/>
  <to uri="bean:validateOrder"/>
  <to uri="mina:tcp://mainframeip:4444?textline=true"/>
  <to uri="bean:sendConfirmEmail"/>
</route>
```

As the `to` is a generic type we must state in the uri scheme which component it is. So we must write `bean:` for the Bean component that we are using.

Conclusion

This example was provided to demonstrate the Spring DSL (XML based) as opposed to the pure Java DSL from the first example. And as well to point about that the `to` doesn't have to be the last node in a route graph.

This example is also based on the **in-only** message exchange pattern. What you must understand as well is the **in-out** message exchange pattern, where the caller expects a response. We will look into this in another example.

See also

- Examples
- Tutorials
- User Guide

CHAPTER 3

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, commonly known as the "Gang of Four" (GoF) book. Since the publication of *Design Patterns*, many other pattern 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 Woolf. It is common for people to refer to this book by its initials *EIP*. As the subtitle of EIP suggests, the book focuses 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, intended to be used in architectural diagrams.

THE CAMEL PROJECT

Camel (<http://camel.apache.org>) is an open-source, Java-based project that helps the user implement many of the design patterns in 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 have the EIP book as a reference.

ONLINE DOCUMENTATION FOR CAMEL

The documentation is all under the Documentation category on the right-side menu of the Camel website (also available in PDF form). Camel-related books are also available, in particular the Camel in Action book, presently serving as the Camel bible—it has a free Chapter One (pdf), which is highly recommended to read to get more familiar with Camel.

A useful tip for navigating the online documentation

The breadcrumbs at the top of the online Camel documentation can help you navigate between parent and child subsections. 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 "Architecture" and "Languages" buttons as indicating you are in the "Languages" section of the "Architecture" chapter. Adding browser bookmarks to pages that you frequently reference can also save time.

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 component 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 and Spring API.

CONCEPTS AND TERMINOLOGY FUNDAMENTAL TO CAMEL

In this section some of the concepts and terminology that are fundamental to Camel are explained. This section is not meant as a complete Camel tutorial, but as 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 – and possibly Components, which are discussed in Section 4.5 ("Components") – 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 – both discussed in Section 4.6 ("Message and Exchange") – to an Endpoint – discussed in Section 4.1 ("Endpoint"). This provides a way to enter messages into source endpoints, so that the messages will move along routes – discussed in Section 4.8 ("Routes, RouteBuilders and Java DSL") – 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 many "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 `c1ass`, 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 message. 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 1. 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 1. 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(RouteBuilder builder)` method invokes `builder.setContext(this)` – so the `RouteBuilder` object knows which `CamelContext` object it is associated with – 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 `RouteBuilder` classes.

Continue Learning about Camel

Return to the main Getting Started page for additional introductory reference information.

CHAPTER 4

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 for using Java for expressions
- Constant
- the unified EL from JSP and JSF
- Header
- JSONPath
- XPath
- Mvel
- OGNL
- Ref Language
- ExchangeProperty / Property
- Scripting Languages such as
 - BeanShell
 - JavaScript
 - Groovy
 - Python
 - PHP
 - Ruby
- Simple
 - File Language
- Spring Expression Language
- SQL
- Tokenizer
- XPath
- XQuery
- VTD-XML

Most of these languages is also supported used as Annotation Based Expression Language.

For a full details of the individual languages see the Language Appendix

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.

important

Make sure to read How do I configure endpoints to learn more about configuring endpoints. For example how to refer to beans in the Registry or how to use raw values for password options, and using property placeholders etc.

Current Supported URIs

Component / ArtifactId / URI	Description
AHC / camel-ahc	To call external HTTP services using Async Http Client ahc: http://hostName[:port][/{resourceUri}][?options]

AHC-WS / camel-ahc-ws	<code>ahc-ws[s]://hostName[:port][/resourceUri][?options]</code>	To exchange data with external Websocket servers using Async Http Client
AMQP / camel-amqp	<code>amqp:[queue:]topic:destinationName[?options]</code>	For Messaging with AMQP protocol
APNS / camel-apns	<code>apns:<notify consumer>[?options]</code>	For sending notifications to Apple iOS devices
Atmosphere-Websocket / camel-atmosphere-websocket	<code>atmosphere-websocket:///relative path[?options]</code>	To exchange data with external Websocket clients using Atmosphere
Atom / camel-atom	<code>atom:atomUri[?options]</code>	Working with Apache Abdera for atom integration, such as consuming an atom feed.
Avro / camel-avro	<code>avro:[transport]:[host]:[port][/messageName][?options]</code>	Working with Apache Avro for data serialization.
AWS-CW / camel-aws	<code>aws-cw://namespace[?options]</code>	For working with Amazon's CloudWatch (CW).
AWS-DDB / camel-aws	<code>aws-ddb://tableName[?options]</code>	For working with Amazon's DynamoDB (DDB).
AWS-DDBSTREAM / camel-aws	<code>aws-ddbstream://tableName[?options]</code>	For working with Amazon's DynamoDB Streams (DDB Streams).
AWS-EC2 / camel-aws	<code>aws-ec2://label[?options]</code>	For working with Amazon's Elastic Compute Cloud (EC2).
AWS-SDB / camel-aws	<code>aws-sdb:/domainName[?options]</code>	For working with Amazon's SimpleDB (SDB).
AWS-SES / camel-aws	<code>aws-ses://from[?options]</code>	For working with Amazon's Simple Email Service (SES).
AWS-SNS / camel-aws	<code>aws-sns://topicName[?options]</code>	For Messaging with Amazon's Simple Notification Service (SNS).
AWS-SQS / camel-aws	<code>aws-sqs://queueName[?options]</code>	For Messaging with Amazon's Simple Queue Service (SQS).
AWS-SWF / camel-aws	<code>aws-swf://<workflow activity>[?options]</code>	For Messaging with Amazon's Simple Workflow Service (SWF).
AWS-S3 / camel-aws		For working with Amazon's Simple Storage Service (S3).

<code>aws-s3://bucketName[?options]</code>	
Bean / camel-core	Uses the Bean Binding to bind message exchanges to beans in the Registry. Is also used for exposing and invoking POJO (Plain Old Java Objects).
<code>bean:beanName[?options]</code>	
Beanstalk / camel-beanstalk	For working with Amazon's Beanstalk.
<code>beanstalk:hostname:port/tube[?options]</code>	
Bean Validator / camel-bean-validator	Validates the payload of a message using the Java Validation API (JSR 303 and JAXP Validation) and its reference implementation Hibernate Validator
<code>bean-validator:label[?options]</code>	
Box / camel-box	For uploading, downloading and managing files, managing files, folders, groups, collaborations, etc. on Box.com.
<code>box://endpoint-prefix/endpoint[?options]</code>	
Braintree / camel-braintree	Component for interacting with Braintree Payments via Braintree Java SDK
<code>braintree://endpoint-prefix/endpoint[?options]</code>	
Browse / camel-core	Provides a simple BrowseableEndpoint which can be useful for testing, visualisation tools or debugging. The exchanges sent to the endpoint are all available to be browsed.
<code>browse:someName</code>	
Cache / camel-cache	The cache component facilitates creation of caching endpoints and processors using EHCache as the cache implementation.
<code>cache://cacheName[?options]</code>	
Cassandra / camel-cassandraql	For integrating with Apache Cassandra.
<code>cql:localhost/keyspace</code>	
Class / camel-core	Uses the Bean Binding to bind message exchanges to beans in the Registry. Is also used for exposing and invoking POJO (Plain Old Java Objects).
<code>class:className[?options]</code>	
Chronicle Engine / camel-chronicle	Chronicle Engine is a high performance, low latency, reactive processing framework.
<code>chronicle-engine:addresses/path[?options]</code>	
Chunk / camel-chunk	Generates a response using a Chunk template
<code>chunk:templateName[?options]</code>	
CMIS / camel-cmis	Uses the Apache Chemistry client API to interface with CMIS supporting CMS
<code>cmis://cmisServerUrl[?options]</code>	
Cometd / camel-cometd	Used to deliver messages using the jetty cometd implementation of the bayeux protocol
<code>cometd://hostName:port/channelName[?options]</code>	
Consul / camel-consul	For interfacing with an Consul.
<code>consul:apiEndpoint[?options]</code>	
Context / camel-context	Used to refer to endpoints within a separate CamelContext to provide a simple black box composition approach so that routes
<code>context:camelContextId:localEndpointName[?options]</code>	

	can be combined into a CamelContext and then used as a black box component inside other routes in other CamelContexts
ControlBus / camel-core	ControlBus EIP that allows to send messages to Endpoints for managing and monitoring your Camel applications.
CouchDB / camel-couchdb	couchdb:hostName[:port]/database[?options] To integrate with Apache CouchDB.
Crypto (Digital Signatures) / camel-crypto	crypto:<sign verify>:name[?options] Used to sign and verify exchanges using the Signature Service of the Java Cryptographic Extension.
CXF / camel-cxf	cxfrs:<bean:cxfEndpoint> //someAddress[?options] Working with Apache CXF for web services integration
CXF Bean / camel-cxf	cxfrs:<bean:serviceBeanRef[?options] Proceess the exchange using a JAX WS or JAX RS annotated bean from the registry. Requires less configuration than the above CXF Component
CXFRS / camel-cxf	cxfrs:<bean:rsEndpoint> //address[?options] Working with Apache CXF for REST services integration
DataFormat / camel-core	dataformat:name:<marshal unmarshal>[?options] for working with Data Formats as if it was a regular Component supporting Endpoints and URLs.
DataSet / camel-core	dataset:name[?options] 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:someName[?options] Synchronous call to another endpoint from same CamelContext.
Direct-VM / camel-core	direct-vm:someName[?options] Synchronous call to another endpoint in another CamelContext running in the same JVM.
DNS / camel-dns	dns:operation[?options] To lookup domain information and run DNS queries using DNSJava
Disruptor / camel-disruptor	disruptor:someName[?<option>] disruptor-vm:someName[?<option>] To provide the implementation of SEDA which is based on disruptor
Docker / camel-docker	docker://[operation]?[options] To communicate with Docker
Dozer / camel-dozer	dozer://name[?options] To convert message body using the Dozer type converter library.
Dropbox / camel-dropbox	dropbox://[operation]?[options] The dropbox: component allows you to treat Dropbox remote folders as a producer or consumer of messages.

EJB / camel-ejb	<code>ejb:ejbName[?options]</code>	Uses the Bean Binding to bind message exchanges to EJBs. It works like the Bean component but just for accessing EJBs. Supports EJB 3.0 onwards.
Ehcache / camel-ehcache	<code>ehcache://cacheName[?options]</code>	The cache component facilitates creation of caching endpoints and processors using Ehcache 3 as the cache implementation.
ElasticSearch / camel-elasticsearch	<code>elasticsearch://clusterName[?options]</code>	For interfacing with an ElasticSearch server.
Etcd / camel-etcd	<code>etcd:namespace[/path][?options]</code>	For interfacing with an Etcd key value store.
Spring Event / camel-spring	<code>spring-event://default</code>	Working with Spring ApplicationEvents
EventAdmin / camel-eventadmin	<code>eventadmin:topic[?options]</code>	Receiving OSGi EventAdmin events
Exec / camel-exec	<code>exec://executable[?options]</code>	For executing system commands
Facebook / camel-facebook	<code>facebook://endpoint[?options]</code>	Providing access to all of the Facebook APIs accessible using Facebook4J
File / camel-core	<code>file://nameOfFileOrDirectory[?options]</code>	Sending messages to a file or polling a file or directory.
Flatpack / camel-flatpack	<code>flatpack:[fixed delim]:configFile[?options]</code>	Processing fixed width or delimited files or messages using the FlatPack library
Flink / camel-flink	<code>flink:dataset[?options]</code> <code>flink:datasream[?options]</code>	Bridges Camel connectors with Apache Flink tasks.
FOP / camel-fop	<code>fop:outputFormat[?options]</code>	Renders the message into different output formats using Apache FOP
FreeMarker / camel-freemarker	<code>freemarker:templateName[?options]</code>	Generates a response using a FreeMarker template
FTP / camel-ftp	<code>ftp:contextPath[?options]</code>	Sending and receiving files over FTP.
FTPS / camel-ftp	<code>ftps://[username@]hostName[:port]/directoryName[?options]</code>	Sending and receiving files over FTP Secure (TLS and SSL).
Ganglia / camel-ganglia	<code>ganglia:destination:port[?options]</code>	Sends values as metrics to the Ganglia performance monitoring system using gmetric4j. Can be used along with JMExtric.

GAuth / camel-gae	<code>gauth://name[?options]</code>	Used by web applications to implement an OAuth consumer. See also Camel Components for Google App Engine.
GHHttp / camel-gae	<code>ghhttp:contextPath[?options]</code>	Provides connectivity to the URL fetch service of Google App Engine but can also be used to receive messages from servlets. See also Camel Components for Google App Engine.
Git / camel-git	<code>git:localRepositoryPath[?options]</code>	Supports interaction with Git repositories
Github / camel-github	<code>github:endpoint[?options]</code>	Supports interaction with Github
GLogin / camel-gae	<code>glogin://hostname[:port][?options]</code>	Used by Camel applications outside Google App Engine (GAE) for programmatic login to GAE applications. See also Camel Components for Google App Engine.
GTask / camel-gae	<code>gtask://queue-name[?options]</code>	Supports asynchronous message processing on Google App Engine by using the task queueing service as message queue. See also Camel Components for Google App Engine.
Google Calendar / camel-google-calendar	<code>google-calendar://endpoint-prefix/endpoint[?options]</code>	Supports interaction with Google Calendar's REST API.
Google Drive / camel-google-drive	<code>google-drive://endpoint-prefix/endpoint[?options]</code>	Supports interaction with Google Drive's REST API.
Google Mail / camel-google-mail	<code>google-mail://endpoint-prefix/endpoint[?options]</code>	Supports interaction with Google Mail's REST API.
GMail / camel-gae	<code>gmail://user@googlemail.com[?options]</code>	Supports sending of emails via the mail service of Google App Engine. See also Camel Components for Google App Engine.
Gora / camel-gora	<code>gora:instanceName[?options]</code>	Supports to work with NoSQL databases using the Apache Gora framework.
Grape/ camel-grape	<code>grape:defaultMavenCoordinates</code>	Grape component allows you to fetch, load and manage additional jars when CamelContext is running.
Geocoder / camel-geocoder	<code>geocoder:<address latlng:latitude,longitude>[?options]</code>	Supports looking up geocoders for an address, or reverse lookup geocoders from an address.
Google Guava EventBus / camel-guava-eventbus	<code>guava-eventbus:busName[?options]</code>	The Google Guava EventBus allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other). This component provides integration bridge between Camel and Google Guava EventBus infrastructure.
Hazelcast / camel-hazelcast	<code>hazelcast://[type]:cachename[?options]</code>	Hazelcast is a data grid entirely implemented in Java (single jar). This component supports map, multimap, seda, queue, set, atomic number and simple cluster support.
HBase / camel-hbase		For reading/writing from/to an HBase store (Hadoop database)

<code>hbase://table[?options]</code>	
HDFS / camel-hdfs	For reading/writing from/to an HDFS filesystem using Hadoop 1.x
<code>hdfs://hostName[:port][/path][?options]</code>	
HDFS2 / camel-hdfs2	For reading/writing from/to an HDFS filesystem using Hadoop 2.x
<code>hdfs2://hostName[:port][/path][?options]</code>	
Hipchat / camel-hipchat	For sending/receiving messages to Hipchat using v2 API
<code>hipchat://[host][:port]?options</code>	
HL7 / camel-hl7	For working with the HL7 MLLP protocol and the HL7 data format using the HAPI library
<code>mina2:tcp://hostName[:port][?options]</code>	
Infinispan / camel-infinispan	For reading/writing from/to Infinispan distributed key/value store and data grid
<code>infinispan://cacheName[?options]</code>	
HTTP / camel-http	For calling out to external HTTP servers using Apache HTTP Client 3.x
<code>http:hostName[:port][/resourceUri][?options]</code>	
HTTP4 / camel-http4	For calling out to external HTTP servers using Apache HTTP Client 4.x
<code>http4:hostName[:port][/resourceUri][?options]</code>	
iBATIS / camel-ibatis	Performs a query, poll, insert, update or delete in a relational database using Apache iBATIS
<code>ibatis://statementName[?options]</code>	
Ignite / camel-ignite	Apache Ignite In-Memory Data Fabric is a high-performance, integrated and distributed in-memory platform for computing and transacting on large-scale data sets in real-time, orders of magnitude faster than possible with traditional disk-based or flash technologies. It is designed to deliver uncompromised performance for a wide set of in-memory computing use cases from high performance computing, to the industry most advanced data grid, highly available service grid, and streaming.
<code>ignite:[cache/compute/messaging/...][?options]</code>	
IMAP / camel-mail	Receiving email using IMAP
<code>imap://[username@]hostName[:port][?options]</code>	
IMAPS / camel-mail	...
<code>imaps://[username@]hostName[:port][?options]</code>	
IRC / camel-irc	For IRC communication
<code>irc:[login@]hostName[:port]/#room[?options]</code>	
IronMQ / camel-ironmq	For working with IronMQ a elastic and durable hosted message queue as a service.
<code>ironmq:queueName[?options]</code>	
JavaSpace / camel-javaspace	Sending and receiving messages through JavaSpace
<code>javaspaces:jini://hostName[?options]</code>	
jBPM / camel-jbpm	Sending messages through kie-remote-client API to jBPM.

<code>jbpm:hostName[:port][/:resourceUri][?options]</code>	
jcache / camel-jcache	The JCache component facilitates creation of caching endpoints and processors using JCache / jsr107 as the cache implementation.
<code>jcache:cacheName[?options]</code>	
jclouds / camel-jclouds	For interacting with cloud compute & blobstore service via jclouds
<code>jclouds:<blobstore compute>:[provider id][?options]</code>	
JCR / camel-jcr	Storing a message in a JCR compliant repository like Apache Jackrabbit
<code>jcr://user:password@repository/path/to/node[?options]</code>	
JDBC / camel-jdbc	For performing JDBC queries and operations
<code>jdbc:dataSourceName[?options]</code>	
Jetty / camel-jetty	For exposing or consuming services over HTTP
<code>jetty:hostName[:port][/:resourceUri][?options]</code>	
JGroups / camel-jgroups	The jgroups: component provides exchange of messages between Camel infrastructure and JGroups clusters.
<code>jgroups:clusterName[?options]</code>	
JIRA / camel-jira	For interacting with JIRA
<code>jira://endpoint[?options]</code>	
JMS / camel-jms	Working with JMS providers
<code>jms:[queue:]topic:destinationName[?options]</code>	
JMX / camel-jmx	For working with JMX notification listeners
<code>jmx://platform[?options]</code>	
JPA / camel-jpa	For using a database as a queue via the JPA specification for working with OpenJPA, Hibernate or TopLink
<code>jpa://entityName[?options]</code>	
JOLT / camel-jolt	The jolt : component allows you to process a JSON messages using an JOLT specification. This can be ideal when doing JSON to JSON transformation.
<code>jolt:specName[?options]</code>	
Jsch / camel-jsch	Support for the scp protocol
<code>scp://hostName[:port]/destination[?options]</code>	
JT/400 / camel-jt400	For integrating with data queues on an AS/400 (aka System i, IBM i, i5, ...) system
<code>jt400://user:pwd@system/<path_to_dtaq>[?options]</code>	
Kafka / camel-kafka	For producing to or consuming from Apache Kafka message brokers.
<code>kafka://server:port[?options]</code>	
Kestrel / camel-kestrel	For producing to or consuming from Kestrel queues
<code>kestrel://[addresslist/]queueName[?options]</code>	

Krati / camel-krati	For producing to or consuming to Krati datastores
<code>krati://[path to datastore/][?options]</code>	
Kubernetes / camel-kubernetes	For integrating your application with Kubernetes standalone or on top of OpenShift.
<code>kubernetes:masterUrl[?options]</code>	
Kura / camel-kura	For deploying Camel OSGi routes into the Eclipse Kura M2M container.
Language / camel-core	Executes Languages scripts
<code>language://languageName[:script][?options]</code>	
LDAP / camel-ldap	Performing searches on LDAP servers (<scope> must be one of object onelevel subtree)
<code>ldap:host[:port][?options]</code>	
LinkedIn / camel-linkedin	Component for retrieving LinkedIn user profiles, connections, companies, groups, posts, etc. using LinkedIn REST API.
<code>linkedin://endpoint-prefix/endpoint[?options]</code>	
Log / camel-core	Uses Jakarta Commons Logging to log the message exchange to some underlying logging system like log4j
<code>log:loggingCategory[?options]</code>	
Lucene / camel-lucene	Uses Apache Lucene to perform Java-based indexing and full text based searches using advanced analysis/tokenization capabilities
<code>lucene:searcherName:<insert query>[?options]</code>	
Lumberjack / camel-lumberjack	Uses the Lumberjack protocol for retrieving logs (from Filebeat for instance)
<code>lumberjack:host[:port]</code>	
Metrics / camel-metrics	Uses Metrics to collect application statistics directly from Camel routes.
<code>metrics:[meter counter histogram timer]:metricname[?options]</code>	
MINA / camel-mina	Working with Apache MINA 1.x
<code>mina:[tcp udp vm]:host[:port][?options]</code>	
MINA2 / camel-mina2	Working with Apache MINA 2.x
<code>mina2:[tcp udp vm]:host[:port][?options]</code>	
Mock / camel-core	For testing routes and mediation rules using mocks
<code>mock:name[?options]</code>	
MLLP / camel-mllp	The MLLP component is specifically designed to handle the nuances of the MLLP protocol and provide the functionality required by Healthcare providers to communicate with other systems using the MLLP protocol
<code>mllp:host:port[?options]</code>	
MongoDB / camel-mongodb	Interacts with MongoDB databases and collections. Offers producer endpoints to perform CRUD-style operations and more against databases and collections, as well as consumer endpoints to listen on collections and dispatch objects to Camel routes
<code>mongodb:connectionBean[?options]</code>	
MongoDB GridFS / camel-mongodb-gridfs	Sending and receiving files via MongoDB's GridFS system. Note: for Camel < 2.19, the URI syntax is gridfs:dbName[?options]
<code>mongodb-gridfs:dbName[?options]</code>	

MQTT / camel-mqtt	Component for communicating with MQTT M2M message brokers
<code>mqtt:name[?options]</code>	
MSV / camel-msv	Validates the payload of a message using the MSV Library
<code>msv:someLocalOrRemoteResource[?options]</code>	
Mustache / camel-mustache	Generates a response using a Mustache template
<code>mustache:templateName[?options]</code>	
MVEL / camel-mvel	Generates a response using an MVEL template
<code>mvel:templateName[?options]</code>	
MyBatis / camel-mybatis	Performs a query, poll, insert, update or delete in a relational database using MyBatis
<code>mybatis:/statementName[?options]</code>	
Nagios / camel-nagios	Sending passive checks to Nagios using JSendNSCA
<code>nagios://hostName[:port][?options]</code>	
NATS / camel-nats	For messaging with the NATS platform.
<code>nats://servers[?options]</code>	
Netty / camel-netty	Working with TCP and UDP protocols using Java NIO based capabilities offered by the Netty project
<code>netty:<tcp udp>/host[:port][?options]</code>	
Netty4 / camel-netty4	Working with TCP and UDP protocols using Java NIO based capabilities offered by the Netty project
<code>netty4:<tcp udp>/host[:port][?options]</code>	
Netty HTTP / camel-netty-http	Netty HTTP server and client using the Netty project
<code>netty-http:http:[port]/context-path[?options]</code>	
Netty4 HTTP / camel-netty4-http	Netty HTTP server and client using the Netty project 4.x
<code>netty4-http:http:[port]/context-path[?options]</code>	
Olingo2 / camel-olingo2	Communicates with OData 2.0 services using Apache Olingo 2.0.
<code>olingo2:endpoint/resource-path[?options]</code>	
Openshift / camel-openshift	To manage your Openshift applications.
<code>openshift:clientId[?options]</code>	
OptaPlanner / camel-optaplanner	Solves the planning problem contained in a message with OptaPlanner.
<code>optaplanner:solverConfig[?options]</code>	
Paho / camel-paho	Paho component provides connector for the MQTT messaging protocol using the Paho library.
<code>paho:topic[?options]</code>	
Pax-Logging / camel-paxlogging	Receiving Pax-Logging events in OSGi
<code>pax-logging:loggerName[?options]</code>	

<code>paxlogging:appender</code>	
PDF / camel-pdf	Allows to work with Apache PDFBox PDF documents
<code>pdf:operation[?options]</code>	
PGEEvent / camel-pgevent	Allows for Producing/Consuming PostgreSQL events related to the LISTEN/NOTIFY commands added since PostgreSQL 8.3
<code>pgevent: dataSource[?options]</code>	
POP3 / camel-mail	Receiving email using POP3 and JavaMail
<code>pop3s://[username@]hostName port[?options]</code>	
POP3S / camel-mail	...
<code>pop3s://[username@]hostName port[?options]</code>	
Printer / camel-printer	The printer component facilitates creation of printer endpoints to local, remote and wireless printers. The endpoints provide the ability to print camel directed payloads when utilized on camel routes.
<code>lpr://host:port/path/to/printer[?options]</code>	
Properties / camel-core	The properties component facilitates using property placeholders directly in endpoint URI definitions.
<code>properties://key[?options]</code>	
Quartz / camel-quartz	Provides a scheduled delivery of messages using the Quartz 1.x scheduler
<code>quartz://groupName/timerName[?options]</code>	
Quartz2 / camel-quartz2	Provides a scheduled delivery of messages using the Quartz 2.x scheduler
<code>quartz2://groupName/timerName[?options]</code>	
Quickfix / camel-quickfix	Implementation of the QuickFix for Java engine which allow to send/receive FIX messages
<code>quickfix:configFile[?options]</code>	
RabbitMQ / camel-rabbitmq	Component for integrating with RabbitMQ
<code>rabbitmq://hostname[:port]/exchangeName[?options]</code>	
Ref / camel-core	Component for lookup of existing endpoints bound in the Registry.
<code>ref:name</code>	
Rest / camel-core	Component for consuming Restful resources supporting the Rest DSL and plugins to other Camel rest components.
<code>rest:verb:path[?options]</code>	
Restlet / camel-restlet	Component for consuming and producing Restful resources using Restlet
<code>restlet:restletUrl[?options]</code>	
REST Swagger / camel-rest-swagger	Component for accessing REST resources using Swagger specification as configuration.
<code>rest-swagger:[specificationUri#]operationId[?options]</code>	
RMI / camel-rmi	Working with RMI
<code>rmi://hostName[:port][?options]</code>	
RNC / camel-jing	Validates the payload of a message using RelaxNG Compact

	Syntax
RNG / camel-jing	<code>rnc:/relativeOrAbsoluteUri[?options]</code> Validates the payload of a message using RelaxNG
Routebox / camel-routebox	<code>routebox:routeBoxName[?options]</code> Facilitates the creation of specialized endpoints that offer encapsulation and a strategy/map based indirection service to a collection of camel routes hosted in an automatically created or user injected camel context
RSS / camel-rss	<code>rss:uri[?options]</code> Working with ROME for RSS integration, such as consuming an RSS feed.
Salesforce / camel-salesforce	<code>salesforce:topic[?options]</code> To integrate with Salesforce
SAP NetWeaver / camel-sap-netweaver	<code>sap-netweaver:hostName[:port][?options]</code> To integrate with SAP NetWeaver Gateway
Scheduler / camel-core	<code>scheduler://name[?options]</code> Used to generate message exchanges when a scheduler fires. The scheduler has more functionality than the timer component.
schematron / camel-schematron	<code>schematron://path[?options]</code> Camel component of Schematron which supports to validate the XML instance documents.
SEDA / camel-core	<code>seda:someName[?options]</code> Asynchronous call to another endpoint in the same CamelContext
ServiceNow / camel-servicenow	<code>servicenow:instanceName[?options]</code> Camel component for ServiceNow
SERVLET / camel-servlet	<code>servlet:relativePath[?options]</code> For exposing services over HTTP through the servlet which is deployed into the Web container.
SFTP / camel-ftp	<code>sftp://[username@]hostName[:port]/directoryName[?options]</code> Sending and receiving files over SFTP (FTP over SSH).
Sip / camel-sip	<code>sip://user@hostName[:port][?options]</code> Publish/Subsribe communication capability using the Telecom SIP protocol. RFC3903 - Session Initiation Protocol (SIP) Extension for Event
SIPS / camel-sip	<code>sips://user@hostName[:port][?options]</code> ...
SJMS / camel-sjms	<code>sjms:[queue:]topic:destinationName[?options]</code> A ground up implementation of a JMS client
SJMS Batch / camel-sjms	<code>sjms-batch:[queue:]destinationName[?options]</code> A specialized JMS component for highly-performant transactional batch consumption from a queue.
Slack / camel-slack	The slack component allows you to connect to an instance of

<code>slack:#channel[?options]</code>	Slack and delivers a message contained in the message body via a pre established Slack incoming webhook .
SMTP / camel-mail <code>smtps://[username@]hostName[:port][?options]</code>	Sending email using SMTP and JavaMail
SMTP / camel-mail <code>smtps://[username@]hostName[:port][?options]</code>	...
SMPP / camel-smpp <code>smpp://[username@]hostName[:port][?options]</code>	To send and receive SMS using Short Messaging Service Center using the JSMPP library
SMPPS / camel-smpp <code>smpps://[username@]hostName[:port][?options]</code>	...
SNMP / camel-snmp <code>snmp://hostName[:port][?options]</code>	Polling OID values and receiving traps using SNMP via SNMP4J library
Solr / camel-solr <code>solr://hostName[:port]/solr[?options]</code>	Uses the Solrj client API to interface with an Apache Lucene Solr server
Apache Spark / camel-spark <code>spark:{rdd dataframe hive}[?options]</code>	Bridges Apache Spark computations with Camel endpoints.
Spark-rest / camel-spark-rest <code>spark-rest://verb:path[?options]</code>	For easily defining REST services endpoints using Spark REST Java library.
Splunk / camel-splunk <code>splunk://[endpoint][?options]</code>	For working with Splunk
SpringBatch / camel-spring-batch <code>spring-batch://jobName[?options]</code>	To bridge Camel and Spring Batch
SpringIntegration / camel-spring-integration <code>spring-integration:defaultChannelName[?options]</code>	The bridge component of Camel and Spring Integration
Spring LDAP / camel-spring-ldap <code>spring-ldap:springLdapTemplateBean[?options]</code>	Camel wrapper for Spring LDAP
Spring Redis / camel-spring-redis <code>spring-redis://hostName:port[?options]</code>	Component for consuming and producing from Redis key-value store Redis
Spring Web Services / camel-spring-ws <code>spring-ws:[mapping-type:]address[?options]</code>	Client-side support for accessing web services, and server-side support for creating your own contract-first web services using Spring Web Services
SQL / camel-sql <code>sql:select * from table where id=#{?options}</code>	Performing SQL queries using JDBC

SQL Stored Procedure / camel-sql	Performing SQL queries using Stored Procedure calls
<code>sql-stored:template[?options]</code>	
SSH component / camel-ssh	For sending commands to a SSH server
<code>ssh:[username[:password]@]hostName[:port][?options]</code>	
StAX / camel-stax	Process messages through a SAX ContentHandler.
<code>stax:(contentHandlerClassName#myHandler)</code>	
Stream / camel-stream	Read or write to an input/output/error/file stream rather like unix pipes
<code>stream:[in out err file header url][?options]</code>	
Stomp / camel-stomp	For communicating with Stomp compliant message brokers, like Apache ActiveMQ or ActiveMQ Apollo
<code>stomp:queue:destinationName[?options]</code>	
StringTemplate / camel-stringtemplate	Generates a response using a String Template
<code>string-template:templateName[?options]</code>	
Stub / camel-core	Allows you to stub out some physical middleware endpoint for easier testing or debugging
<code>stub:someOtherCamelUri[?options]</code>	
Telegram / camel-telegram	Allows to exchange data with the Telegram messaging network
<code>telegram://bots/authToken[?options]</code>	
Test / camel-spring	Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint
<code>test:expectedMessagesEndpointUri[?options]</code>	
Timer / camel-core	Used to generate message exchanges when a timer fires You can only consume events from this endpoint.
<code>timer:timerName[?options]</code>	
Twitter / camel-twitter	A twitter endpoint
<code>twitter://endpoint[?options]</code>	
Undertow / camel-undertow	HTTP server and client using the light-weight Undertow server.
<code>undertow://host:port/context-path[?options]</code>	
Validation / camel-core (camel-spring for Camel 2.8 or older)	Validates the payload of a message using XML Schema and JAXP Validation
<code>validation:someLocalOrRemoteResource[?options]</code>	
Velocity / camel-velocity	Generates a response using an Apache Velocity template
<code>velocity:templateName[?options]</code>	
Vertx / camel-vertx	Working with the vertx event bus
<code>vertx:eventBusName</code>	
VM / camel-core	Asynchronous call to another endpoint in the same JVM

<code>vm:queueName[?options]</code>	
Weather / camel-weather	Polls the weather information from Open Weather Map
<code>wweather://name[?options]</code>	
Websocket / camel-websocket	Communicating with Websocket clients
<code>websocket://hostname[:port]/resourceUri[?options]</code>	
XML Security / camel-xmlsecurity	Used to sign and verify exchanges using the XML signature specification.
<code>xmlsecurity:<sign verify>:name[?options]</code>	
XMPP / camel-xmpp	Working with XMPP and Jabber
<code>xmpp://login@]hostname[:port][/participant][?options]</code>	
XQuery / camel-saxon	Generates a response using an XQuery template
<code>xquery:someXQueryResource</code>	
XSLT / camel-core (camel-spring for Camel 2.8 or older)	Generates a response using an XSLT template
<code>xslt:templateName[?options]</code>	
Yammer / camel-yammer	Allows you to interact with the Yammer enterprise social network
<code>yammer://function[?options]</code>	
Zookeeper / camel-zookeeper	Working with ZooKeeper cluster(s)
<code>zookeeper://zookeeperServer[:port][/path][?options]</code>	

URI's for external components

Other projects and companies have also created Camel components to integrate additional functionality into Camel. These components may be provided under licenses that are not compatible with the Apache License, use libraries that are not compatible, etc... These components are not supported by the Camel team, but we provide links here to help users find the additional functionality.

Component / ArtifactId / URI	License	Description
ActiveMQ / activemq-camel	Apache	For JMS Messaging with Apache ActiveMQ.
<code>activemq:[queue topic:]destinationName</code>		
ActiveMQ Broker / activemq-camel	Apache	For internal message routing in the ActiveMQ broker using Camel.
<code>broker:[queue topic:]destinationName</code>		
Activiti / activiti-camel	Apache	For working with Activiti, a light-weight workflow and Business Process Management (BPM) platform which supports BPMN 2.
<code>activiti:camelProcess:serviceTask</code>		
Bluetooth / camel-bluetooth / rhiot.io	Apache	Camel Bluetooth component can

<code>bluetooth:label</code>			retrieve information about the Bluetooth devices available within the device range.
<code>Couchbase / camel-couchbase / camel-extra</code>	<code>couchbase:protocol://host[:port]/bucket</code>	Couchbase	Working with Couchbase NoSQL document database.
<code>Db4o / camel-db4o / camel-extra</code>	<code>db4o://className</code>	GPL	For using a db4o datastore as a queue via the db4o library.
<code>Esper / camel-esper / camel-extra</code>	<code>esper:name</code>	GPL	Working with the Esper Library for Event Stream Processing.
<code>Fabric AMQ / mq-fabric-camel / fabric8</code>	<code>amq:[queue topic:]destinationName</code>	Apache	The amq : endpoint works exactly like the activemq : endpoint in Apache Camel; only it uses the fabric to automatically discover the broker. So there is no configuration required; it'll just work out of the box and automatically discover whatever ActiveMQ message brokers are available; with failover and load balancing.
<code>Fabric Fabric / fabric-camel / fabric8</code>	<code>fabric:logicalName:camelEndpointUri</code>	Apache	The fabric : endpoint uses Fabric's discovery mechanism to expose physical sockets, HTTP endpoints, etc. into the runtime registry using a logical name so that clients can use the existing Camel Load Balancer.
<code>Fabric Master / fabric-camel / fabric8</code>	<code>master:clusterName:camelEndpointUri</code>	Apache	The master : endpoint provides a way to ensure only a single consumer in a cluster consumes from a given endpoint; with automatic failover if that JVM dies.
<code>Framebuffer / camel-framebuffer / rhiot.io</code>	<code>framebuffer://name</code>	Apache	Camel Framebuffer component can be used to manage any Linux Framebuffer.
<code>gpsd / camel-gpsd / rhiot.io</code>	<code>gpsd:label[?options]</code>	Apache	Camel GPSD component can be used to read current GPS information from GPS devices.
<code>Hibernate / camel-hibernate / camel-extra</code>	<code>hibernate://entityName</code>	GPL	For using a database as a queue via the Hibernate library.
<code>JBI / servicemix-camel</code>	<code>jbi:serviceName</code>	Apache	For JBI integration such as working with Apache ServiceMix.
<code>JCIFS / camel-jcifs / camel-extra</code>	<code>smb://user@server.example.com/sharename?password=secret&localWorkDirectory=/tmp</code>	LGPL	This component provides access to remote file systems over the CIFS/SMB networking protocol by using the JCIFS library.
<code>kura-cloud / camel-kura / rhiot.io</code>	<code>kura-wifi:networkInterface/ssid</code>	Apache	Camel Kura Cloud component interacts directly with Kura CloudService.
<code>kura-wifi / camel-kura / rhiot.io</code>	<code>kura-wifi:networkInterface/ssid</code>	Apache	Camel Kura WiFi component can be used to retrieve the information about the WiFi access

			spots available within the device range.
NMR / servicemix-nmr	<code>nmr://serviceName</code>	Apache	Integration with the Normalized Message Router BUS in ServiceMix 4.x.
OpenIMAJ / camel-openimaj / rhiot.io	<code>pi4j-gpio://gpioId[?options]</code>	Apache	Camel OpenIMAJ component can be used to detect faces in images.
pi4j-gpio / camel-pi4j / rhiot.io	<code>pi4j-gpio://gpioId[?options]</code>	Apache	GPIO Component for RaspberryPi based on pi4j lib.
pi4j-i2c / camel-pi4j / rhiot.io	<code>pi4j-i2c://busId/deviceId[?options]</code>	Apache	i2c Component for RaspberryPi based on pi4j lib.
PubNub / camel-pubnub / rhiot.io	<code>pubnub://pubnubEndpointType:channel[?options]</code>	Apache	Camel PubNub component. More information rhiot.io project.
RCode / camel-rcode / camel-extra	<code>rcode://host[:port]/operation[?options]</code>	LGPL	Uses Rserve to integrate Camel with the statistics environment R.
Scalate / scalate-camel	<code>scalate:templateName</code>	Apache	Uses the given Scalate template to transform the message.
Smooks / camel-smooks / camel-extra	<code>unmarshal(edt)</code>	GPL	For working with EDI parsing using the Smooks library. This component is deprecated as Smooks now provides Camel integration out of the box.
Spring Neo4j / camel-spring-neo4j / camel-extra	<code>spring-neo4j:http://hostname[:port]/database[?options]</code>	TBA	Component for producing to Neo4j datastore using the Spring Data Neo4j library.
Tinkerforge / camel-tinkerforge / rhiot.io	<code>tinkerforge://hostname[:port]]/devicetype/uid/[?options]</code>	Apache	The tinkerforge component allows interaction with Tinkerforge bricklets. It uses the standard Java bindings to connects to brickd. For more information see the rhiot.io.
VirtualBox / camel-virtualbox / camel-extra	<code>virtualbox:machine[?options]</code>	GPL V2	The VirtualBox component uses the webservice API that exposes VirtualBox functionality and consumes events generated by virtual machines.
Webcam / camel-webcam / rhiot.io	<code>webcam:label[?options]</code>	Apache	Camel Webcam component can be used to capture still images and detect motion.
ZeroMQ / camel-zeromq / camel-extra	<code>zeromq:(tcp ipc)://hostname:port</code>	LGPL	The ZeroMQ component allows you to consumer or produce messages using ZeroMQ.

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

CHAPTER 5**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

	Event Message	How can messaging be used to transmit events from one application to another?
	Request Reply	When an application sends a message, how can it get a response from the receiver?
	Correlation Identifier	How does a requestor that has received a reply know which request this is the reply for?
	Return Address	How does a replier know where to send the reply?

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?
	Dynamic Router	How can you avoid the dependency of the router on all possible destinations while maintaining its efficiency?
	Recipient List	How do we route a message to a list of (static or 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?
	Composed Message Processor	How can you maintain the overall message flow when processing a message consisting of multiple elements, each of which may require different processing?
	Scatter-Gather	How do you maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply?
	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?
	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?
	Sampling	How can I sample one message out of many in a given period to avoid downstream route does not get overloaded?
	Delay	How can I delay the sending of a message?
	Load Balancer	How can I balance load across a number of endpoints?
	Hystrix	To use Hystrix Circuit Breaker when calling an external service.
	Service Call	To call a remote service in a distributed system where the service is looked up from a service registry of some sorts.
	Multicast	How can I route a message to a number of endpoints at the same time?
	Loop	How can I repeat processing a message in a loop?

Message Transformation

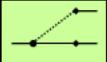
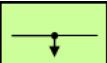
	Content Enricher	How do we communicate with another system if the message originator does not have all the required data items available?
	Content Filter	How do you simplify dealing with a large message, when you are interested only in a few data items?
	Claim Check	How can we reduce the data volume of message sent across the system without sacrificing information content?
	Normalizer	How do you process messages that are semantically equivalent, but arrive in a different format?
	Sort	How can I sort the body of a message?
	Script	How do I execute a script which may not change the message?
	Validate	How can I validate a message?

Messaging Endpoints

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

System Management

	ControlBus	How can we effectively administer a messaging system that is distributed across multiple platforms and a wide geographic area?
--	------------	--

	Detour	How can you route a message through intermediate steps to perform validation, testing or debugging functions?
	Wire Tap	How do you inspect messages that travel on a point-to-point channel?
	Message History	How can we effectively analyze and debug the flow of messages in a loosely coupled system?
	Log	How can I log processing a message?

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

CookBook

This document describes various recipes for working with Camel

- Bean Integration describes how to work with beans and Camel in a loosely coupled way so that your beans do not have to depend on any Camel APIs
 - Annotation Based Expression Language binds expressions to method parameters
 - Bean Binding defines which methods are invoked and how the Message is converted into the parameters of the method when it is invoked
 - Bean Injection for injecting Camel related resources into your POJOs
 - Parameter Binding Annotations for extracting various headers, properties or payloads from a Message
 - POJO Consuming for consuming and possibly routing messages from Camel
 - POJO Producing for producing camel messages from your POJOs
 - RecipientList Annotation for creating a Recipient List from a POJO method
 - Using Exchange Pattern Annotations describes how pattern annotations can be used to change the behaviour of method invocations
- Hiding Middleware describes how to avoid your business logic being coupled to any particular middleware APIs allowing you to easily switch from in JVM SEDA to JMS, ActiveMQ, Hibernate, JPA, JDBC, iBatis or JavaSpace etc.
- Visualisation describes how to visualise your Enterprise Integration Patterns to help you understand your routing rules
- Business Activity Monitoring (BAM) for monitoring business processes across systems
- Extract Transform Load (ETL) to load data into systems or databases
- Testing for testing distributed and asynchronous systems using a messaging approach
 - Camel Test for creating test cases using a single Java class for all your configuration and routing
 - Spring Testing uses Spring Test together with either XML or Java Config to dependency inject your test classes
 - Guice uses Guice to dependency inject your test classes
- Templating is a great way to create service stubs to be able to test your system without some back end system.
- Database for working with databases
- Parallel Processing and Ordering on how using parallel processing and SEDA or JMS based load balancing can be achieved.
- Asynchronous Processing in Camel Routes.
- Implementing Virtual Topics on other JMS providers shows how to get the effect of Virtual Topics and avoid issues with JMS durable topics
- Camel Transport for CXF describes how to put the Camel context into the CXF transport layer.
- Fine Grained Control Over a Channel describes how to deliver a sequence of messages over a single channel and then stopping any more messages being sent over that channel. Typically used for sending data over a socket and then closing the socket.
- EventNotifier to log details about all sent Exchanges shows how to let Camels EventNotifier log all sent to endpoint events and how long time it took.
- Loading routes from XML files into an existing CamelContext.
- Using MDC logging with Camel
- Running Camel standalone and have it keep running shows how to keep Camel running when you run it standalone.
- Hazelcast Idempotent Repository Tutorial shows how to avoid to consume duplicated messages in a clustered environment.
- How to use Camel as a HTTP proxy between a client and server shows how to use Camel as a HTTP adapter/proxy between a client and HTTP service.

BEAN INTEGRATION

Camel supports the integration of beans and POJOs in a number of ways

Annotations

If a bean is defined in Spring XML or scanned using the Spring component scanning mechanism and a `<camelContext>` is used or a `CamelBeanPostProcessor` then we process a number of Camel annotations to do various things such as injecting resources or producing, consuming or routing messages.

The following annotations is supported and inject by Camel's `CamelBeanPostProcessor`

Annotation	Description
<code>@EndpointInject</code>	To inject an endpoint, see more details at POJO Producing.

<code>@BeanInject</code>	Camel 2.13: To inject a bean obtained from the Registry. See Bean Injection.
<code>@PropertyInject</code>	Camel 2.12: To inject a value using property placeholder.
<code>@Produce</code>	To inject a producer to send message to an endpoint. See POJO Producing.
<code>@Consume</code>	To inject a consumer on a method. See POJO Consuming.

See more details at:

- POJO Consuming to consume and possibly route messages from Camel
- POJO Producing to make it easy to produce camel messages from your POJOs
- DynamicRouter Annotation for creating a Dynamic Router from a POJO method
- RecipientList Annotation for creating a Recipient List from a POJO method
- RoutingSlip Annotation for creating a Routing Slip for a POJO method
- Bean Injection to inject Camel related resources into your POJOs
- Using Exchange Pattern Annotations describes how the pattern annotations can be used to change the behaviour of method invocations with Spring Remoting or POJO Producing

Example

See the POJO Messaging Example for how to use the annotations for routing and messaging.

Bean Component

The Bean component allows one to invoke a particular method. Alternately the Bean component supports the creation of a proxy via ProxyHelper to a Java interface; which the implementation just sends a message containing a BeanInvocation to some Camel endpoint.

Spring Remoting

We support a Spring Remoting provider which uses Camel as the underlying transport mechanism. The nice thing about this approach is we can use any of the Camel transport Components to communicate between beans. It also means we can use Content Based Router and the other Enterprise Integration Patterns in between the beans; in particular we can use Message Translator to be able to convert what the on-the-wire messages look like in addition to adding various headers and so forth.

Bean binding

Whenever Camel invokes a bean method via one of the above methods (Bean component, Spring Remoting or POJO Consuming) then the **Bean Binding** mechanism is used to figure out what method to use (if it is not explicit) and how to bind the Message to the parameters possibly using the Parameter Binding Annotations or using a method name option.

Annotation Based Expression Language

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
<code>@Bean</code>	Inject a Bean expression
<code>@BeanShell</code>	Inject a BeanShell expression
<code>@Constant</code>	Inject a Constant expression
<code>@EL</code>	Inject an EL expression
<code>@Groovy</code>	Inject a Groovy expression
<code>@Header</code>	Inject a Header expression
<code>@JavaScript</code>	Inject a JavaScript expression
<code>@MVEL</code>	Inject a MVEL expression
<code>@OGNL</code>	Inject an OGNL expression
<code>@PHP</code>	Inject a PHP expression
<code>@Python</code>	Inject a Python expression
<code>@Ruby</code>	Inject a Ruby expression
<code>@Simple</code>	Inject an Simple expression
<code>@XPath</code>	Inject an XPath expression
<code>@XQuery</code>	Inject an XQuery expression

Example:

```
public class Foo {
    @MessageDriven(uri = "activemq:my.queue")
```

```
public void doSomething(@XPath("/foo/bar/text()") String correlationID, @Body String body) {
    // process the inbound message here
}
```

Advanced example using @Bean

And an example of using the the @Bean binding annotation, where you can use a POJO where you can do whatever java code you like:

```
public class Foo {
    @MessageDriven(uri = "activemq:my.queue")
    public void doSomething(@Bean("myCorrelationIdGenerator") String correlationID, @Body String body) {
        // process the inbound message here
    }
}
```

And then we can have a spring bean with the id **myCorrelationIdGenerator** where we can compute the id.

```
public class MyIdGenerator {
    private UserManager userManager;

    public String generate(@Header("name = "user") String user, @Body String payload) throws Exception {
        User user = userManager.lookupUser(user);
        String userId = user.getPrimaryId();
        String id = userId + generateHashCodeForPayload(payload);
        return id;
    }
}
```

The POJO MyIdGenerator has one public method that accepts two parameters. However we have also annotated this one with the @Header and @Body annotation to help Camel know what to bind here from the Message from the Exchange being processed.

Of course this could be simplified a lot if you for instance just have a simple id generator. But we wanted to demonstrate that you can use the Bean Binding annotations anywhere.

```
public class MySimpleIdGenerator {
    public static int generate() {
        // generate a unique id
        return 123;
    }
}
```

And finally we just need to remember to have our bean registered in the Spring Registry:

```
<bean id="myCorrelationIdGenerator" class="com.mycompany.MySimpleIdGenerator"/>
```

Example using Groovy

In this example we have an Exchange that has a User object stored in the in header. This User object has methods to get some user information. We want to use Groovy to inject an expression that extracts and concats the fullname of the user into the fullName parameter.

```
public void doSomething(@Groovy("$request.header[user].firstName $request.header[user].familyName") String fullName, @Body
    // process the inbound message here
)
```

Groovy supports GStrings that is like a template where we can insert \$ placeholders that will be evaluated by Groovy.

BEAN BINDING

Bean Binding in Camel defines both which methods are invoked and also how the Message is converted into the parameters of the method when it is invoked.

Choosing the method to invoke

The binding of a Camel Message to a bean method call can occur in different ways, in the following order of importance:

- if the message contains the header **CamelBeanMethodName** then that method is invoked, converting the body to the type of the method's argument.
 - From **Camel 2.8** onwards you can qualify parameter types to select exactly which method to use among overloads with the same name (see below for more details).

- From **Camel 2.9** onwards you can specify parameter values directly in the method option (see below for more details).
- you can explicitly specify the method name in the DSL or when using POJO Consuming or POJO Producing
- if the bean has a method marked with the @Handler annotation, then that method is selected
- if the bean can be converted to a Processor using the Type Converter mechanism, then this is used to process the message. The ActiveMQ component uses this mechanism to allow any JMS MessageListener to be invoked directly by Camel without having to write any integration glue code. You can use the same mechanism to integrate Camel into any other messaging/remoting frameworks.
- if the body of the message can be converted to a BeanInvocation (the default payload used by the ProxyHelper) component - then that is used to invoke the method and pass its arguments
- otherwise the type of the body is used to find a matching method; 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.
- if the bean class is private (or package-private), interface methods will be preferred (from **Camel 2.9** onwards) since Camel can't invoke class methods on such beans

In cases where Camel cannot choose a method to invoke, an `AmbiguousMethodCallException` is thrown.

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

Asynchronous processing

From **Camel 2.18** onwards you can return a CompletionStage implementation (e.g. a `CompletableFuture`) to implement asynchronous processing.

Please be sure to properly complete the CompletionStage with the result or exception, including any timeout handling. Exchange processing would wait for completion and would not impose any timeouts automatically. It's extremely useful to monitor Inflight repository for any hanging messages.

Note that completing with "null" won't set outbound message body to null, but would keep message intact. This is useful to support methods that don't modify exchange and return `CompletableFuture<Void>`. To set body to null, just add Exchange method parameter and directly modify exchange messages.

Examples:

Simple asynchronous processor, modifying message body.

```
javapublic CompletableFuture<String> doSomethingAsync(String body)
```

Composite processor that do not modify exchange

```
javapublic CompletableFuture<Void> doSomethingAsync(String body) { return CompletableFuture.allOf(doA(body), doB(body), doC()); }
```

Parameter binding

When a method has been chosen for invocation, Camel will bind to the parameters of the method.

The following Camel-specific types are automatically bound:

- `org.apache.camel.Exchange`
- `org.apache.camel.Message`
- `org.apache.camel.CamelContext`
- `org.apache.camel.TypeConverter`
- `org.apache.camel.spi.Registry`
- `java.lang.Exception`

So, if you declare any of these types, they will be provided by Camel. **Note that Exception will bind to the caught exception of the Exchange** - so it's often usable if you employ a Pojo to handle, e.g., an `onException` route.

What is most interesting is that Camel will also try to bind the body of the Exchange to the first parameter of the method signature (albeit not of any of the types above). So if, for instance, we declare a parameter as `String body`, then Camel will bind the IN body to this type. Camel will also automatically convert to the type declared in the method signature.

Let's review some examples:

Below is a simple method with a body binding. Camel will bind the IN body to the `body` parameter and convert it to a `String`.

```
public String doSomething(String body)
```

In the following sample we got one of the automatically-bound types as well - for instance, a `Registry` that we can use to lookup beans.

```
public String doSomething(String body, Registry registry)
```

We can use Exchange as well:

```
public String doSomething(String body, Exchange exchange)
```

You also have multiple types:

```
public String doSomething(String body, Exchange exchange, TypeConverter converter)
```

And imagine you use a Pojo to handle a given custom exception `InvalidOrderException` - we can then bind that as well:

```
public String badOrder(String body, InvalidOrderException invalid)
```

Notice that we can bind to it even if we use a sub type of `java.lang.Exception` as Camel still knows it's an exception and can bind the cause (if any exists).

So what about headers and other stuff? Well now it gets a bit tricky - so we can use annotations to help us, or specify the binding in the method name option.
See the following sections for more detail.

Binding Annotations

You can use the Parameter Binding Annotations to customize how parameter values are created from the Message

Examples

For example, a Bean 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** when there is only a single parameter of the type org.apache.camel.Exchange:

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

@Handler

You can mark a method in your bean with the **@Handler** annotation to indicate that this method should be used for Bean Binding. This has an advantage as you need not specify a method name in the Camel route, and therefore do not run into problems after renaming the method in an IDE that can't find all its references.

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

Parameter binding using method option

Available as of Camel 2.9

Camel uses the following rules to determine if it's a parameter value in the method option

- The value is either `true` or `false` which denotes a boolean value
- The value is a numeric value such as `123` or `7`
- The value is a String enclosed with either single or double quotes
- The value is `null` which denotes a `null` value
- It can be evaluated using the Simple language, which means you can use, e.g., `body`, `header.foo` and other Simple tokens. Notice the tokens must be enclosed with `{}.`.

Any other value is consider to be a type declaration instead - see the next section about specifying types for overloaded methods.

When invoking a Bean you can instruct Camel to invoke a specific method by providing the method name:

```
.bean(OrderService.class, "doSomething")
```

Here we tell Camel to invoke the `doSomething` method - Camel handles the parameters' binding. Now suppose the method has 2 parameters, and the 2nd parameter is a boolean where we want to pass in a true value:

```
public void doSomething(String payload, boolean highPriority) { ... }
```

This is now possible in **Camel 2.9** onwards:

```
.bean(OrderService.class, "doSomething(*, true)")
```

In the example above, we defined the first parameter using the wild card symbol `*`, which tells Camel to bind this parameter to any type, and let Camel figure this out. The 2nd parameter has a fixed value of `true`. Instead of the wildcard symbol we can instruct Camel to use the message body as shown:

```
.bean(OrderService.class, "doSomething(${body}, true)")
```

The syntax of the parameters is using the Simple expression language so we have to use `{}.` placeholders in the body to refer to the message body.

If you want to pass in a `null` value, then you can explicit define this in the method option as shown below:

```
.to("bean:orderService?method=doSomething(null, true)")
```

Specifying `null` as a parameter value instructs Camel to force passing a `null` value.

Besides the message body, you can pass in the message headers as a `java.util.Map`:

```
.bean(OrderService.class, "doSomethingWithHeaders(${body}, ${headers})")
```

You can also pass in other fixed values besides booleans. For example, you can pass in a String and an integer:

```
.bean(MyBean.class, "echo('World', 5)")
```

In the example above, we invoke the `echo` method with two parameters. The first has the content 'World' (without quotes), and the 2nd has the value of 5.

Camel will automatically convert these values to the parameters' types.

Having the power of the Simple language allows us to bind to message headers and other values such as:

```
.bean(OrderService.class, "doSomething(${body}, ${header.high})")
```

You can also use the OGNL support of the Simple expression language. Now suppose the message body is an object which has a method named `asXml1`. To invoke the `asXml1` method we can do as follows:

```
.bean(OrderService.class, "doSomething(${body.asXml}, ${header.high}))")
```

Instead of using `.bean` as shown in the examples above, you may want to use `.to` instead as shown:

```
.to("bean:orderService?method=doSomething(${body.asXml}, ${header.high})")
```

Using type qualifiers to select among overloaded methods

Available as of Camel 2.8

If you have a Bean with overloaded methods, you can now specify parameter types in the method name so Camel can match the method you intend to use.

Given the following bean:{snippet:id=e1|lang=java|title=MyBean|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/bean/BeanOverloadedMethodTest.java}Then the MyBean has 2 overloaded methods with the names hello and times. So if we want to use the method which has 2 parameters we can do as follows in the Camel route:{snippet:id=e2|lang=java|title=Invoke 2 parameter method|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/bean/BeanOverloadedMethodTest.java}We can also use a * as wildcard so we can just say we want to execute the method with 2 parameters we do:{snippet:id=e3|lang=java|title=Invoke 2 parameter method using wildcard|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/bean/BeanOverloadedMethodTest.java}By default Camel will match the type name using the simple name, e.g. any leading package name will be disregarded. However if you want to match using the FQN, then specify the FQN type and Camel will leverage that. So if you have a com.foo.MyOrder and you want to match against the FQN, and **not** the simple name "MyOrder", then follow this example:

```
.bean(OrderService.class, "doSomething(com.foo.MyOrder)")
```

Camel currently only supports either specifying parameter binding or type per parameter in the method name option. You **cannot** specify both at the same time, such as

```
doSomething(com.foo.MyOrder ${body}, boolean ${header.high})
```

This may change in the future.

Bean Injection

We support the injection of various resources using @EndpointInject or @BeanInject. This can be used to inject

- Endpoint instances which can be used for testing when used with Mock endpoints; see the Spring Testing for an example.
- ProducerTemplate instances for POJO Producing
- client side proxies for POJO Producing which is a simple approach to Spring Remoting

Using @BeanInject

From Camel 2.13 onwards you can inject beans (obtained from the Registry) into your beans such as RouteBuilder classes.

For example to inject a bean named foo, you can enlist the bean in the Registry such as in a Spring XML file:

```
<bean id="foo" class="com.foo.MyFooBean"/>
```

And then in a Java RouteBuilder class, you can inject the bean using @BeanInject as shown below:

```
public class MyRouteBuilder extends RouteBuilder {
    @BeanInject("foo")
    MyFooBean foo;
    public void configure() throws Exception {
        ...
    }
}
```

If you omit the name, then Camel does a lookup by type, and injects the bean if there is exactly one bean of that type enlisted in the Registry.

```
@BeanInject
MyFooBean foo;
```

Parameter Binding Annotations

camel-core

The annotations below are all part of **camel-core** and thus does not require **camel-spring** or Spring. These annotations can be used with the Bean component or when invoking beans in the DSL

Annotations can be used to define an Expression or to extract various headers, properties or payloads from a Message when invoking a bean method (see Bean Integration for more detail of how to invoke bean methods) together with being useful to help disambiguate which method to invoke.

If no annotations are used then Camel assumes that a single parameter is the body of the message. Camel will then use the Type Converter mechanism to convert from the expression value to the actual type of the parameter.

The core annotations are as follows

Annotation	Meaning	Parameter
@Body	To bind to an inbound message body	

@ExchangeException	To bind to an Exception set on the exchange	
@Header	To bind to an inbound message header	String name of the header
@Headers	To bind to the Map of the inbound message headers	
@OutHeaders	To bind to the Map of the outbound message headers	
@Property	To bind to a named property on the exchange	String name of the property
@Properties	To bind to the property map on the exchange	
@Handler	Not part as a type parameter but stated in this table anyway to spread the good word that we have this annotation in Camel now. See more at Bean Binding.	

The follow annotations @Headers, @OutHeaders and @Properties binds to the backing `java.util.Map` so you can alter the content of these maps directly, for instance using the `put` method to add a new entry. See the `OrderService` class at `Exception Clause` for such an example. You can use `@Header("myHeader")` and `@Property("myProperty")` to access the backing `java.util.Map`.

Example

In this example below we have a `@Consume` consumer (like message driven) that consumes JMS messages from the activemq queue. We use the `@Header` and `@Body` parameter binding annotations to bind from the `JMSMessage` to the method parameters.

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

In the above Camel will extract the value of `Message.getJMSCorrelationID()`, then using the Type Converter to adapt the value to the type of the parameter if required - it will inject the parameter value for the `correlationID` parameter. Then the payload of the message will be converted to a String and injected into the `body` parameter.

You don't necessarily need to use the `@Consume` annotation if you don't want to as you could also make use of the Camel DSL to route to the bean's method as well.

Using the DSL to invoke the bean method

Here is another example which does not use POJO Consuming annotations but instead uses the DSL to route messages to the bean method

```
public class Foo {
    public void doSomething(@Header("JMSCorrelationID") String correlationID, @Body String body) {
        // process the inbound message here
    }
}
```

The routing DSL then looks like this

```
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("user") String user, @Body String body, Exchange exchange) {
    exchange.getIn().setBody(body + "MyBean");
}
```

Annotation Based Expression Language

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
@Bean	Inject a Bean expression
@BeanShell	Inject a BeanShell expression
@Constant	Inject a Constant expression
@EL	Inject an EL expression
@Groovy	Inject a Groovy expression
@Header	Inject a Header expression
@JavaScript	Inject a JavaScript expression
@MVEL	Inject a MVEL 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

Example:

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

Advanced example using @Bean

And an example of using the the @Bean binding annotation, where you can use a POJO where you can do whatever java code you like:

```
public class Foo {
    @MessageDriven(uri = "activemq:my.queue")
    public void doSomething(@Bean("myCorrelationIdGenerator") String correlationID, @Body String body) {
        // process the inbound message here
    }
}
```

And then we can have a spring bean with the id `myCorrelationIdGenerator` where we can compute the id.

```
public class MyIdGenerator {
    private UserManager userManager;

    public String generate(@Header("name = "user") String user, @Body String payload) throws Exception {
        User user = userManager.lookupUser(user);
        String userId = user.getPrimaryId();
        String id = userId + generateHashCodeForPayload(payload);
        return id;
    }
}
```

The POJO `MyIdGenerator` has one public method that accepts two parameters. However we have also annotated this one with the `@Header` and `@Body` annotation to help Camel know what to bind here from the Message from the Exchange being processed.

Of course this could be simplified a lot if you for instance just have a simple id generator. But we wanted to demonstrate that you can use the Bean Binding annotations anywhere.

```
public class MySimpleIdGenerator {
    public static int generate() {
        // generate a unique id
        return 123;
    }
}
```

And finally we just need to remember to have our bean registered in the Spring Registry:

```
<bean id="myCorrelationIdGenerator" class="com.mycompany.MySimpleIdGenerator"/>
```

Example using Groovy

In this example we have an Exchange that has a User object stored in the in header. This User object has methods to get some user information. We want to use Groovy to inject an expression that extracts and concats the fullname of the user into the fullName parameter.

```
public void doSomething(@Groovy("${request.header[user].firstName ${request.header[user].familyName}) String fullName, @Body
} // process the inbound message here
}
```

Groovy supports GStrings that is like a template where we can insert \$ placeholders that will be evaluated by Groovy.

@Consume

To consume a message you use the `@Consume` annotation to mark a particular method of a bean as being a consumer method. The uri of the annotation defines the Camel Endpoint to consume from.

e.g. lets invoke the `oncheese()` method with the String body of the inbound JMS message from ActiveMQ on the cheese queue; this will use the Type Converter to convert the JMS ObjectMessage or BytesMessage to a String - or just use a TextMessage from JMS

```
public class Foo {
    @Consume(uri="activemq:cheese")
    public void onCheese(String name) {
        ...
    }
}
```

The Bean Binding is then used to convert the inbound Message to the parameter list used to invoke the method .

What this does is basically create a route that looks kinda like this

```
from(uri).bean(theBean, "methodName");
```

When using more than one CamelContext

When you use more than 1 CamelContext you might end up with each of them creating a POJO Consuming; therefore use the option context on `@Consume` that allows you to specify which CamelContext id/name you want it to apply for.

Using context option to apply only a certain CamelContext

See the warning above.

You can use the context option to specify which CamelContext the consumer should only apply for. For example:

```
@Consume(uri="activemq:cheese", context="camel-1")
public void onCheese(String name) {
```

The consumer above will only be created for the CamelContext that have the context id = camel-1. You set this id in the XML tag:

```
<camelContext id="camel-1" ...>
```

Using an explicit route

If you want to invoke a bean method from many different endpoints or within different complex routes in different circumstances you can just use the normal routing DSL or the Spring XML configuration file.

For example

```
from(uri).beanRef("myBean", "methodName");
```

which will then look up in the Registry and find the bean and invoke the given bean name. (You can omit the method name and have Camel figure out the right method based on the method annotations and body type).

Use the Bean endpoint

You can always use the bean endpoint

```
from(uri).to("bean:myBean?method=methodName");
```

Using a property to define the endpoint

Available as of Camel 2.11

The following annotations @Consume, @Produce, @EndpointInject, now offers a property attribute you can use to define the endpoint as a property on the bean. Then Camel will use the getter method to access the property.

This applies for them all

The explanation below applies for all the three annotations, eg @Consume, @Produce, and @EndpointInject

For example

```
public class MyService {
    private String serviceEndpoint;

    public void setServiceEndpoint(String uri) {
        this.serviceEndpoint = uri;
    }

    public String getServiceEndpoint() {
        return serviceEndpoint;
    }

    @Consume(property = "serviceEndpoint")
    public void onService(String input) {
        ...
    }
}
```

The bean MyService has a property named serviceEndpoint which has getter/setter for the property. Now we want to use the bean for POJO Consuming, and hence why we use @Consume in the onService method. Notice how we use the property = "serviceEndpoint" to configure the property that has the endpoint url.

If you define the bean in Spring XML or Blueprint, then you can configure the property as follows:

```
<bean id="myService" class="com.foo.MyService">
    <property name="serviceEndpoint" value="activemq:queue:foo"/>
</bean>
```

This allows you to configure the bean using any standard IoC style.

Camel offers a naming convention which allows you to not have to explicit name the property.

Camel uses this algorithm to find the getter method. The method must be a getXXX method.

1. Use the property name if explicit given
2. If no property name was configured, then use the method name
3. Try to get the property with name*Endpoint* (eg with Endpoint as postfix)
4. Try to get the property with the name as is (eg no postfix or postfix)
5. If the property name starts with **on** then omit that, and try step 3 and 4 again.

So in the example above, we could have defined the @Consume annotation as

```
@Consume(property = "service")
public void onService(String input) {
```

Now the property is named 'service' which then would match step 3 from the algorithm, and have Camel invoke the getServiceEndpoint method.

We could also have omitted the property attribute, to make it implicit

```
@Consume
public void onService(String input) {
```

Now Camel matches step 5, and loses the prefix **on** in the name, and looks for 'service' as the property. And because there is a getServiceEndpoint method, Camel will use that.

Which approach to use?

Using the @Consume annotations are simpler when you are creating a simple route with a single well defined input URI.

However if you require more complex routes or the same bean method needs to be invoked from many places then please use the routing DSL as shown above.

There are two different ways to send messages to any Camel Endpoint from a POJO

@EndpointInject

To allow sending of messages from POJOs you can use the @EndpointInject annotation. This will inject a ProducerTemplate so that the bean can participate in message exchanges.

Example: send a message to the foo.bar ActiveMQ queue:

```
public class Foo {
    @EndpointInject(uri="activemq:foo.bar")
    ProducerTemplate producer;

    public void doSomething() {
        if (whatever) {
            producer.sendBody("<hello>world!</hello>");
        }
    }
}
```

```

    }
}
```

The downside of this is that your code is now dependent on a Camel API, the `ProducerTemplate`. The next section describes how to remove this dependency.

See POJO Consuming for how to use a property on the bean as endpoint configuration, e.g., using the `property` attribute on `@Produce`, `@EndpointInject`.

Hiding the Camel APIs From Your Code Using `@Produce`

We recommend Hiding Middleware APIs from your application code so the next option might be more suitable. You can add the `@Produce` annotation to an injection point (a field or property setter) using a `ProducerTemplate` or using some interface you use in your business logic. Example:

```

public interface MyListener {
    String sayHello(String name);
}

public class MyBean {
    @Produce(uri = "activemq:foo")
    protected MyListener producer;

    public void doSomething() {
        // lets send a message
        String response = producer.sayHello("James");
    }
}
```

Here Camel will automatically inject a smart client side proxy at the `@Produce` annotation - an instance of the `MyListener` instance. When we invoke methods on this interface the method call is turned into an object and using the Camel Spring Remoting mechanism it is sent to the endpoint - in this case the ActiveMQ endpoint to queue `foo`; then the caller blocks for a response.

If you want to make asynchronous message sends then use an `@InOnly` annotation on the injection point.

`@RECIPIENTLIST ANNOTATION`

We support the use of `@RecipientList` on a bean method to easily create a dynamic Recipient List using a Java method.

Simple Example using `@Consume` and `@RecipientList`

```

package com.acme.foo;

public class RouterBean {
    @Consume(uri = "activemq:foo")
    @RecipientList
    public String[] route(String body) {
        return new String[]{"activemq:bar", "activemq:whatnot"};
    }
}
```

For example if the above bean is configured in Spring when using a `<camelContext>` element as follows

```

<?xml version="1.0" encoding="UTF-8"?>
<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.xsd
                           http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
                           ">

    <camelContext xmlns="http://activemq.apache.org/camel/schema/spring"/>
    <bean id="myRecipientList" class="com.acme.foo.RouterBean"/>

</beans>
```

then a route will be created consuming from the `foo` queue on the ActiveMQ component which when a message is received the message will be forwarded to the endpoints defined by the result of this method call - namely the `bar` and `whatnot` queues.

How it works

The return value of the `@RecipientList` method is converted to either a `java.util.Collection` / `java.util.Iterator` or array of objects where each element is converted to an Endpoint or a String, or if you are only going to route to a single endpoint then just return either an Endpoint object or an object that can be converted to a String. So the following methods are all valid

```

@RecipientList
public String[] route(String body) { ... }

@RecipientList
public List<String> route(String body) { ... }
```

```

@RecipientList
public Endpoint route(String body) { ... }

@RecipientList
public Endpoint[] route(String body) { ... }

@RecipientList
public Collection<Endpoint> route(String body) { ... }

@RecipientList
public URI route(String body) { ... }

@RecipientList
public URI[] route(String body) { ... }

```

Then for each endpoint or URI the message is forwarded a separate copy to that endpoint.

You can then use whatever Java code you wish to figure out what endpoints to route to; for example you can use the Bean Binding annotations to inject parts of the message body or headers or use Expression values on the message.

More Complex Example Using DSL

In this example we will use more complex Bean Binding, plus we will use a separate route to invoke the Recipient List

```

public class RouterBean2 {

    @RecipientList
    public String route(@Header("customerID") String custID String body) {
        if (custID == null) return null;
        return "activemq:Customers.Orders." + custID;
    }

    public class MyRouteBuilder extends RouteBuilder {
        protected void configure() {
            from("activemq:Orders.Incoming").recipientList(bean("myRouterBean", "route"));
        }
    }
}

```

Notice how we are injecting some headers or expressions and using them to determine the recipients using Recipient List EIP. See the Bean Integration for more details.

USING EXCHANGE PATTERN ANNOTATIONS

When working with POJO Producing or Spring Remoting you invoke methods which typically by default are InOut for Request Reply. That is there is an In message and an Out for the result. Typically invoking this operation will be synchronous, the caller will block until the server returns a result.

Camel has flexible Exchange Pattern support - so you can also support the Event Message pattern to use InOnly for asynchronous or one way operations. These are often called 'fire and forget' like sending a JMS message but not waiting for any response.

From 1.5 onwards Camel supports annotations for specifying the message exchange pattern on regular Java methods, classes or interfaces.

Specifying InOnly methods

Typically the default InOut is what most folks want but you can customize to use InOnly using an annotation.

```

public interface Foo {
    Object someInOutMethod(String input);
    String anotherInOutMethod(Cheese input);

    @InOnly
    void someInOnlyMethod(Document input);
}

```

The above code shows three methods on an interface; the first two use the default InOut mechanism but the `someInOnlyMethod` uses the InOnly annotation to specify it as being a oneway method call.

Class level annotations

You can also use class level annotations to default all methods in an interface to some pattern such as

```

@InOnly
public interface Foo {
    void someInOnlyMethod(Document input);
    void anotherInOnlyMethod(String input);
}

```

Annotations will also be detected on base classes or interfaces. So for example if you created a client side proxy for

```

public class MyFoo implements Foo {
    ...
}

```

```
}
```

Then the methods inherited from Foo would be InOnly.

Overloading a class level annotation

You can overload a class level annotation on specific methods. A common use case for this is if you have a class or interface with many InOnly methods but you want to just annotate one or two methods as InOut

```
@InOnly
public interface Foo {
    void someInOnlyMethod(Document input);
    void anotherInOnlyMethod(String input);

    @InOut
    String someInOutMethod(String input);
}
```

In the above Foo interface the **someInOutMethod** will be InOut

Using your own annotations

You might want to create your own annotations to represent a group of different bits of metadata; such as combining synchrony, concurrency and transaction behaviour.

So you could annotate your annotation with the @Pattern annotation to default the exchange pattern you wish to use.

For example lets say we want to create our own annotation called @MyAsyncService

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})

// lets add the message exchange pattern to it
@Pattern(ExchangePattern.InOnly)

// lets add some other annotations - maybe transaction behaviour?

public @interface MyAsyncService { }
```

Now we can use this annotation and Camel will figure out the correct exchange pattern...

```
public interface Foo {
    void someInOnlyMethod(Document input);
    void anotherInOnlyMethod(String input);

    @MyAsyncService
    String someInOutMethod(String input);
}
```

When writing software these days, its important to try and decouple as much middleware code from your business logic as possible.

This provides a number of benefits...

- you can choose the right middleware solution for your deployment and switch at any time
- you don't have to spend a large amount of time learning the specifics of any particular technology, whether its JMS or JavaSpace or Hibernate or JPA or iBatis whatever

For example if you want to implement some kind of message passing, remoting, reliable load balancing or asynchronous processing in your application we recommend you use Camel annotations to bind your services and business logic to Camel Components which means you can then easily switch between things like

- in JVM messaging with SEDA
- using JMS via ActiveMQ or other JMS providers for reliable load balancing, grid or publish and subscribe
- for low volume, but easier administration since you're probably already using a database you could use
 - Hibernate or JPA to use an entity bean / table as a queue
 - iBatis to work with SQL
 - JDBC for raw SQL access
- use JavaSpace

How to decouple from middleware APIs

The best approach when using remoting is to use Spring Remoting which can then use any messaging or remoting technology under the covers. When using Camel's implementation you can then use any of the Camel Components along with any of the Enterprise Integration Patterns.

Another approach is to bind Java beans to Camel endpoints via the Bean Integration. For example using POJO Consuming and POJO Producing you can avoid using any Camel APIs to decouple your code both from middleware APIs *and* Camel APIs!



VISUALISATION

This functionality is deprecated and to be removed in future Camel releases.

Camel supports the visualisation of your Enterprise Integration Patterns using the GraphViz DOT files which can either be rendered directly via a suitable GraphViz tool or turned into HTML, PNG or SVG files via the Camel Maven Plugin.

Here is a typical example of the kind of thing we can generate



If you click on the actual generated html you will see that you can navigate from an EIP node to its pattern page, along with getting hover-over tool tips etc.

How to generate

See Camel Dot Maven Goal or the other maven goals Camel Maven Plugin

For OS X users

If you are using OS X then you can open the DOT file using graphviz which will then automatically re-render if it changes, so you end up with a real time graphical representation of the topic and queue hierarchies!

Also if you want to edit the layout a little before adding it to a wiki to distribute to your team, open the DOT file with OmniGraffle then just edit away 😊

BUSINESS ACTIVITY MONITORING

The **Camel BAM** module provides a Business Activity Monitoring (BAM) framework for testing business processes across multiple message exchanges on different Endpoint instances.

Consider, for example, a simple system in which you submit Purchase Orders into system A and then receive Invoices from system B. You might want to test that, for a given Purchase Order, you receive a matching Invoice from system B within a specific time period.

How Camel BAM Works

Camel BAM uses a Correlation Identifier on an input message to determine the *Process Instance* to which it belongs. The process instance is an entity bean which can maintain state for each *Activity* (where an activity typically maps to a single endpoint - such as the submission of Purchase Orders or the receipt of Invoices).

You can then add rules to be triggered when a message is received on any activity - such as to set time expectations or perform real time reconciliation of values across activities.

Simple Example

The following example shows how to perform some time based rules on a simple business process of 2 activities - A and B - which correspond with Purchase Orders and Invoices in the example above. If you would like to experiment with this scenario, you may edit this Test Case, which defines the activities and rules, and then tests that they work.

```
{snippet:id=example|lang=java|url=camel/trunk/components/camel-bam/src/test/java/org/apache/camel/bam/BamRouteTest.java}
As you can see in the above example, we first define two activities, and then rules to specify when we expect them to complete for a process instance and when an error condition should be raised.p. The ProcessBuilder is a RouteBuilder and can be added to any CamelContext.
```

Complete Example

For a complete example please see the BAM Example, which is part of the standard Camel Examples

Use Cases

In the world of finance, a common requirement is tracking trades. Often a trader will submit a Front Office Trade which then flows through the Middle Office and Back Office through various systems to settle the trade so that money is exchanged. You may wish to test that the front and back office trades match up within a certain time period; if they don't match or a back office trade does not arrive within a required amount of time, you might signal an alarm.

EXTRACT TRANSFORM LOAD (ETL)

The ETL (Extract, Transform, Load) is a mechanism for loading data into systems or databases using some kind of Data Format from a variety of sources; often files then using Pipes and Filters, Message Translator and possibly other Enterprise Integration Patterns.

So you could query data from various Camel Components such as File, HTTP or JPA, perform multiple patterns such as Splitter or Message Translator then send the messages to some other Component.

To show how this all fits together, try the ETL Example

MOCK COMPONENT

Testing Summary Include

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 run, which typically fires messages to one or more endpoints, and finally the expectations can be asserted in a test case to ensure the system worked as expected.

This allows you to test various things like:

- The correct number of messages are received on each endpoint,
- The correct payloads are received, in the right order,
- Messages arrive on an endpoint in order, using some Expression to create an order testing function,
- Messages arrive match some kind of Predicate such as that specific headers have certain values, or that parts of the messages match some predicate, such as by evaluating an XPath or XQuery Expression.

Note that there is also the Test endpoint which is a Mock endpoint, but which uses a second endpoint to provide the list of expected message bodies and automatically sets up the Mock endpoint assertions. In other words, it's a Mock endpoint that automatically sets up its assertions from some sample messages in a File or database, for example.

Mock endpoints keep received Exchanges in memory indefinitely.

Remember that Mock is designed for testing. When you add Mock endpoints to a route, each Exchange sent to the endpoint will be stored (to allow for later validation) in memory until explicitly reset or the JVM is restarted. If you are sending high volume and/or large messages, this may cause excessive memory use. If your goal is to test deployable routes inline, consider using NotifyBuilder or AdviceWith in your tests instead of adding Mock endpoints to routes directly.

From Camel 2.10 onwards there are two new options retainFirst, and retainLast that can be used to limit the number of messages the Mock endpoints keep in memory.

URI format

mock:someName[?options]

Where **someName** can be any string that uniquely identifies the endpoint.

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

Options

confluenceTableSmall

Option	Default	Description
reportGroup	null	A size to use a throughput logger for reporting
retainFirst		Camel 2.10: To only keep first X number of messages in memory.
retainLast		Camel 2.10: To only keep last X number of messages in memory.

Simple Example

Here's a simple example of Mock endpoint in use. First, the endpoint is resolved on the context. Then we set an expectation, and then, after the test has run, we assert that our expectations have been met.

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class);
resultEndpoint.expectedMessageCount(2); // send some messages ... // now lets assert that the mock:foo endpoint received 2 messages
resultEndpoint.assertIsSatisfied();
```

You typically always call the assertIsSatisfied() method to test that the expectations were met after running a test.

Camel will by default wait 10 seconds when the assertIsSatisfied() is invoked. This can be configured by setting the setResultWaitTime(milliseconds) method.

Using assertPeriod

Available as of Camel 2.7

When the assertion is satisfied then Camel will stop waiting and continue from the assertIsSatisfied method. That means if a new message arrives on the mock endpoint, just a bit later, that arrival will not affect the outcome of the assertion. Suppose you do want to test that no new messages arrives after a period thereafter, then you can do that by setting the setAssertPeriod method, for example:

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class); resultEndpoint.setAssertPeriod(5000);
resultEndpoint.expectedMessageCount(2); // send some messages ... // now lets assert that the mock:foo endpoint received 2 messages
resultEndpoint.assertIsSatisfied();
```

Setting expectations

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

confluenceTableSmall

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).
expectedHeaderReceived(...)	To define the expected header that should be received
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 <code>JMSMessageID</code> 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 the following code:

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

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

Mocking existing endpoints

Available as of Camel 2.7

Camel now allows you to automatically mock existing endpoints in your Camel routes.

How it works

Important: The endpoints are still in action. What happens differently is that a Mock endpoint is injected and receives the message first and then delegates the message to the target endpoint. You can view this as a kind of intercept and delegate or endpoint listener.

Suppose you have the given route below:

```
{snippet:id=route|title=Route|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/interceptor/AdviceWithMockEndpointsTest.java}
```

You can then use the `adviceWith` feature in Camel to mock all the endpoints in a given route from your unit test, as shown below:

```
{snippet:id=e1|title=adviceWith mocking all endpoints|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/interceptor/AdviceWithMockEndpointsTest.java}
```

Notice that the mock endpoints is given the uri `mock:<endpoint>`, for example `mock:direct:foo`. Camel logs at `INFO` level the endpoints being mocked:

`INFO` Advised endpoint [direct://foo] with mock endpoint [mock:direct:foo] Mocked endpoints are without parameters

Endpoints which are mocked will have their parameters stripped off. For example the endpoint "`log:foo?showAll=true`" will be mocked to the following endpoint "`mock:log:foo`". Notice the parameters have been removed.

Its also possible to only mock certain endpoints using a pattern. For example to mock all `log` endpoints you do as shown:

```
{snippet:id=e2|lang=java|title=adviceWith mocking only log endpoints using a pattern|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/interceptor/AdviceWithMockEndpointsTest.java}
```

The pattern supported can be a wildcard or a regular expression. See more details about this at Intercept as its the same matching function used by Camel.

Mind that mocking endpoints causes the messages to be copied when they arrive on the mock.

That means Camel will use more memory. This may not be suitable when you send in a lot of messages.

Mocking existing endpoints using the camel-test component

Instead of using the `adviceWith` to instruct Camel to mock endpoints, you can easily enable this behavior when using the `camel-test` Test Kit.

The same route can be tested as follows. Notice that we return "*" from the `isMockEndpoints` method, which tells Camel to mock all endpoints.

If you only want to mock all `log` endpoints you can return "`log*`" instead.

```
{snippet:id=e1|lang=java|title=isMockEndpoints using camel-test kit|url=camel/trunk/components/camel-test/src/test/java/org/apache/camel/test/patterns/isMockEndpointsJUnit4Test.java}
```

Mocking existing endpoints with XML DSL

If you do not use the `camel-test` component for unit testing (as shown above) you can use a different approach when using XML files for routes.

The solution is to create a new XML file used by the unit test and then include the intended XML file which has the route you want to test.

Suppose we have the route in the `camel-route.xml` file:

```
{snippet:id=e1|lang=xml|title=camel-route.xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/mock/camel-route.xml}
```

Then we create a new XML file as follows, where we include the `camel-route.xml` file and define a spring bean with the class `org.apache.camel.impl.InterceptSendToMockEndpointStrategy` which tells Camel to mock all endpoints:

```
{snippet:id=e1|lang=xml|title=test-camel-route.xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/mock/InterceptSendToMockEndpointStrategyTest.xml}
```

Then in your unit test you load the new XML file (`test-camel-route.xml`) instead of `camel-route.xml`.

To only mock all Log endpoints you can define the pattern in the constructor for the bean:

```
xml-bean id="mockAllEndpoints" class="org.apache.camel.impl.InterceptSendToMockEndpointStrategy">> <constructor-arg index="0" value="log"/> </bean>
```

Mocking endpoints and skip sending to original endpoint

Available as of Camel 2.10

Sometimes you want to easily mock and skip sending to a certain endpoints. So the message is detoured and send to the mock endpoint only. From Camel 2.10 onwards you can now use the `mockEndpointsAndSkip` method using `AdviceWith` or the Test Kit. The example below will skip sending to the two endpoints "`direct:foo`", and "`direct:bar`".

```
{snippet:id=e1|lang=java|title=adviceWith mock and skip sending to endpoints|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/interceptor/AdviceWithMockMultipleEndpointsWithSkipTest.java}
```

The same example using the Test Kit

```
{snippet:id=e1|lang=java|title=isMockEndpointsAndSkip using camel-test kit|url=camel/trunk/components/camel-test/src/test/java/org/apache/camel/test/patterns/IsMockEndpointsAndSkipJUnit4Test.java}
```

Limiting the number of messages to keep

Available as of Camel 2.10

The Mock endpoints will by default keep a copy of every Exchange that it received. So if you test with a lot of messages, then it will consume memory.

From Camel 2.10 onwards we have introduced two options `retainFirst` and `retainLast` that can be used to specify to only keep N'th of the first and/or last Exchanges.

For example in the code below, we only want to retain a copy of the first 5 and last 5 Exchanges the mock receives.

```
MockEndpoint mock = getMockEndpoint("mock:data"); mock.setRetainFirst(5); mock.setRetainLast(5);  
mock.expectedMessageCount(2000); ... mock.assertIsSatisfied();
```

Using this has some limitations. The `getExchanges()` and `getReceivedExchanges()` methods on the `MockEndpoint` will return only the retained copies of the Exchanges. So in the example above, the list will contain 10 Exchanges; the first five, and the last five.

The `retainFirst` and `retainLast` options also have limitations on which expectation methods you can use. For example the `expectedXXX` methods that work on message bodies, headers, etc. will only operate on the retained messages. In the example above they can test only the expectations on the 10 retained messages.

Testing with arrival times

Available as of Camel 2.7

The Mock endpoint stores the arrival time of the message as a property on the Exchange.

```
Date time = exchange.getProperty(Exchange.RECEIVED_TIMESTAMP, Date.class);
```

You can use this information to know when the message arrived on the mock. But it also provides foundation to know the time interval between the previous and next message arrived on the mock. You can use this to set expectations using the `arrives` DSL on the Mock endpoint.

For example to say that the first message should arrive between 0-2 seconds before the next you can do:

```
mock.message(0).arrives().noLaterThan(2).seconds().beforeNext();
```

You can also define this as that 2nd message (0 index based) should arrive no later than 0-2 seconds after the previous:

```
mock.message(1).arrives().noLaterThan(2).seconds().afterPrevious();
```

You can also use `between` to set a lower bound. For example suppose that it should be between 1-4 seconds:

```
mock.message(1).arrives().between(1, 4).seconds().afterPrevious();
```

You can also set the expectation on all messages, for example to say that the gap between them should be at most 1 second:

```
mock.allMessages().arrives().noLaterThan(1).seconds().beforeNext(); time units
```

In the example above we use `seconds` as the time unit, but Camel offers `milliseconds`, and `minutes` as well.

Endpoint See Also

- Spring Testing
- Testing

TESTING

Testing is a crucial activity in any piece of software development or integration. Typically Camel Riders use various different technologies wired together in a variety of patterns with different expression languages together with different forms of Bean Integration and Dependency Injection so its very easy for things to go wrong! 😊 . Testing is the crucial weapon to ensure that things work as you would expect.

Camel is a Java library so you can easily wire up tests in whatever unit testing framework you use (JUnit 3.x (deprecated), 4.x, or TestNG). However the Camel project has tried to make the testing of Camel as easy and powerful as possible so we have introduced the following features.

Testing Mechanisms

The following mechanisms are supported:

Name	Component	Description
Camel Test	camel-test	Is a standalone Java library letting you easily create Camel test cases using a single Java class for all your configuration and routing without using CDI, Spring or Guice for Dependency Injection which does not require an in-depth knowledge of Spring + Spring Test or Guice. Supports JUnit 3.x (deprecated) and JUnit 4.x based tests.
CDI Testing	camel-test-cdi	Provides a JUnit 4 runner that bootstraps a test environment using CDI so that you don't have to be familiar with any CDI testing frameworks and can concentrate on the testing logic of your Camel CDI applications. Testing frameworks like Arquillian or PAX Exam, can be used for more advanced test cases, where you need to configure your system under test in a very fine-grained way or target specific CDI containers.
Spring Testing	camel-test-spring	Supports JUnit 3.x (deprecated) or JUnit 4.x based tests that bootstrap a test environment using Spring without needing to be familiar with Spring Test. The plain JUnit 3.x/4.x based tests work very similar to the test support classes in camel-test.
		Also supports Spring Test based tests that use the declarative style of test configuration and injection common in Spring Test. The Spring Test based tests provide feature parity with the plain JUnit 3.x/4.x based testing approach. Note: camel-test-spring is a new component from Camel 2.10 . For older Camel release use camel-test which has built-in Spring Testing.
Blueprint Testing	camel-test-blueprint	Camel 2.10: Provides the ability to do unit testing on blueprint configurations
Guice	camel-guice	Deprecated Uses Guice to dependency inject your test classes
Camel TestNG	camel-testng	Deprecated Supports plain TestNG based tests with or without CDI, Spring or Guice for Dependency Injection which does not require an in-depth knowledge of CDI, Spring + Spring Test or Guice. From Camel 2.10 : this component supports Spring Test based tests that use the declarative style of test configuration and injection common in Spring Test and described in more detail under Spring Testing.

In all approaches the test classes look pretty much the same in that they all reuse the Camel binding and injection annotations.

Camel Test Example

Here is the Camel Test example:{snippet:lang=java|id=example|url=camel/trunk/components/camel-test/src/test/java/org/apache/camel/test/patterns/FilterTest.java}Notice how it derives from the Camel helper class CamelTestSupport but has no CDI, Spring or Guice dependency injection configuration but instead overrides the createRouteBuilder() method.

CDI Test Example

Here is the CDI Testing example:{snippet:lang=java|id=example|url=camel/trunk/components/camel-test-cdi/src/test/java/org/apache/camel/test/cdi/FilterTest.java}You can find more testing patterns illustrated in the camel-example-cdi-test example and the test classes that come with it.

Spring Test with XML Config Example

Here is the Spring Testing example using XML Config:{snippet:lang=java|id=example|url=camel/trunk/components/camel-spring/src/test/java/org/apache/camel/spring/patterns/FilterTest.java}Notice that we use @MockBean on the test methods to force Spring Testing to automatically reload the CamelContext after each test method - this ensures that the tests don't clash with each other, e.g., one test method sending to an endpoint that is then reused in another test method.

Also note the use of @ContextConfiguration to indicate that by default we should look for the FilterTest-context.xml on the classpath to configure the test case which looks like this:{snippet:lang=xml|id=example|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/patterns/FilterTest-context.xml}

Spring Test with Java Config Example

Here is the Spring Testing example using Java Config.

For more information see Spring Java Config.{snippet:lang=java|id=example|url=camel/trunk/components/camel-spring-javaconfig/src/test/java/org/apache/camel/spring/javaconfig/patterns/FilterTest.java}This is similar to the XML Config example above except that there is no XML file and instead the nested `contextConfig` class does all of the configuration; so your entire test case is contained in a single Java class. We currently have to reference by class name this class in the `@ContextConfiguration` which is a bit ugly. Please vote for SJC-238 to address this and make Spring Test work more cleanly with Spring JavaConfig.

Its totally optional but for the `contextConfig` implementation we derive from `SingleRouteCamelConfiguration` which is a helper Spring Java Config class which will configure the `CamelContext` for us and then register the `RouteBuilder` we create.

Since Camel 2.11.0 you can use the `CamelSpringJUnit4ClassRunner` with `CamelSpringDelegatingTestContextLoader` like example using Java Config with `CamelSpringJUnit4ClassRunner`:

```
{snippet:lang=java|id=example|url=camel/trunk/components/camel-spring-javaconfig/src/test/java/org/apache/camel/spring/javaconfig/test/CamelSpringDelegatingTestContextLoaderTest.java}
```

Spring Test with XML Config and Declarative Configuration Example

Here is a Camel test support enhanced Spring Testing example using XML Config and pure Spring Test based configuration of the Camel Context.{snippet:lang=java|id=1|url=camel/trunk/components/camel-test-spring/src/test/java/org/apache/camel/test/spring/CamelSpringJUnit4ClassRunnerPlainTest.java}Notice how a custom test runner is used with the `@RunWith` annotation to support the features of `CamelTestSupport` through annotations on the test class. See Spring Testing for a list of annotations you can use in your tests.

Blueprint Test

Here is the Blueprint Testing example using XML Config:{snippet:lang=java|id=example|url=camel/trunk/components/camel-test-blueprint/src/test/java/org/apache/camel/test/blueprint/DebugBlueprintTest.java}Also notice the use of `getBlueprintDescriptors` to indicate that by default we should look for the `camelContext.xml` in the package to configure the test case which looks like this:{snippet:lang=xml|id=example|url=camel/trunk/components/camel-test-blueprint/src/test/resources/org/apache/camel/test/blueprint/camelContext.xml}

Testing Endpoints

Camel provides a number of endpoints which can make testing easier.

Name	Description
DataSet	For load & soak testing this endpoint provides a way to create huge numbers of messages for sending to Components and asserting that they are consumed correctly
Mock	For testing routes and mediation rules using mocks and allowing assertions to be added to an endpoint
Test	Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint

The main endpoint is the Mock endpoint which allows expectations to be added to different endpoints; you can then run your tests and assert that your expectations are met at the end.

Stubbing out physical transport technologies

If you wish to test out a route but want to avoid actually using a real physical transport (for example to unit test a transformation route rather than performing a full integration test) then the following endpoints can be useful.

Name	Description
Direct	Direct invocation of the consumer from the producer so that single threaded (non-SEDA) in VM invocation is performed which can be useful to mock out physical transports
SEDA	Delivers messages asynchronously to consumers via a <code>java.util.concurrent.BlockingQueue</code> which is good for testing asynchronous transports
Stub	Works like SEDA but does not validate the endpoint URI, which makes stubbing much easier.

Testing existing routes

Camel provides some features to aid during testing of existing routes where you cannot or will not use Mock etc. For example you may have a production ready route which you want to test with some 3rd party API which sends messages into this route.

Name	Description
NotifyBuilder	Allows you to be notified when a certain condition has occurred. For example when the route has completed five messages. You can build complex expressions to match your criteria when to be notified.
AdviceWith	Allows you to advice or enhance an existing route using a <code>RouteBuilder</code> style. For example you can add

interceptors to intercept sending outgoing messages to assert those messages are as expected.

CAMEL TEST

As a simple alternative to using CDI Testing, Spring Testing or Guice the **camel-test** module was introduced so you can perform powerful Testing of your Enterprise Integration Patterns easily.

JUnit or TestNG

The camel-test JAR is using JUnit. There is an alternative camel-testng JAR (from **Camel 2.8**) using the TestNG test framework.

Adding to your pom.xml

To get started using Camel Test you will need to add an entry to your pom.xml:

JUnit

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

TestNG

Available as of Camel 2.8

```
xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-testng</artifactId> <version>${camel-version}</version> <scope>test</scope> </dependency>
```

You might also want to add slf4j and log4j to ensure nice logging messages (and maybe adding a log4j.properties file into your src/test/resources directory).

```
xml<dependency> <groupId>org.slf4j</groupId> <artifactId>slf4j-log4j12</artifactId> <scope>test</scope> </dependency>
<dependency> <groupId>log4j</groupId> <artifactId>log4j</artifactId> <scope>test</scope> </dependency>
```

Writing your test

You firstly need to derive from the class `CamelTestSupport` (`org.apache.camel.test.CamelTestSupport`, `org.apache.camel.test.junit4.CamelTestSupport`, or `org.apache.camel.testng.CamelTestSupport` for JUnit 3.x, JUnit 4.x, and TestNG, respectively) and typically you will need to override the `createRouteBuilder()` or `createRouteBuilders()` method to create routes to be tested.

Here is an example.{snippet:lang=java:id=example:url=camel/trunk/components/camel-test/src/test/java/org/apache/camel/test/patterns/FilterTest.java}Note: how you can use the various Camel binding and injection annotations to inject individual Endpoint objects - particularly the Mock endpoints which are very useful for Testing. Also you can inject producer objects such as ProducerTemplate or some application code interface for sending messages or invoking services.

Features Provided by CamelTestSupport

The various `CamelTestSupport` classes provide a standard set of behaviors relating to the `CamelContext` used to host the route(s) under test. The classes provide a number of methods that allow a test to alter the configuration of the `CamelContext` used. The following table describes the available customization methods and the default behavior of tests that are built from a `CamelTestSupport` class.

Method Name	Description
<code>boolean isUseRouteBuilder()</code>	If the route builders returned from either <code>createRouteBuilder()</code> or <code>createRouteBuilders()</code> should be added to the <code>CamelContext</code> for the test to be started.
<code>boolean isUseAdviceWith()</code>	If the <code>CamelContext</code> used in the test should be automatically started before test methods are invoked.
<code>boolean isCreateCamelContextPerClass()</code>	Override when using advice with and return true. This helps in knowing the <code>adviceWith()</code> is to be used, and the <code>CamelContext</code> will not be started before the advice with takes place. This delay helps by ensuring the advice with has been properly setup before the <code>CamelContext</code> is started. Its important to start the <code>CamelContext</code> manually from the unit test after you are done doing all the advice with.
	See Setup CamelContext once per class, or per every test method.

<code>String isMockEndpoints()</code>	Triggers the auto-mocking of endpoints whose URIs match the provided filter. The default filter is null which disables this feature. Return "*" to match all endpoints. See <code>org.apache.camel.impl.InterceptSendToMockEndpointStrategy</code> for more details on the registration of the mock endpoints.
<code>boolean isUseDebugger()</code>	If this method returns true, the methods:
	<ul style="list-style-type: none"> • <code>debugBefore(Exchange exchange, Processor processor, ProcessorDefinition<?> definition, String id, String label)</code> • <code>debugAfter(Exchange exchange, Processor processor, ProcessorDefinition<?> definition, String id, String label, long timeTaken)</code>
	are invoked for each processor in the registered routes.
<code>int getShutdownTimeout()</code>	Returns the number of seconds that Camel should wait for graceful shutdown. Useful for decreasing test times when a message is still in flight at the end of the test.
<code>boolean useJmx()</code>	If JMX should be disabled on the CamelContext used in the test.
<code>JndiRegistry createRegistry()</code>	Provides a hook for adding objects into the registry. Override this method bind objects to the registry before test methods are invoked.
<code>useOverridePropertiesWithPropertiesComponent</code>	Camel 2.10: Allows to add/override properties when Using PropertyPlaceholder in Camel.
<code>ignoreMissingLocationWithPropertiesComponent</code>	Camel 2.10: Allows to control if Camel should ignore missing locations for properties.
<code>boolean isDumpRouteCoverage</code>	Camel 2.16: If enabled, then Camel will dump all route coverage statistics into XML files in the target/camel-route-coverage directory. These XML files contains information about "route coverage" of all the routes that was used during the unit test. This allows tooling to inspect these XML files and generate nice route coverage reports.

JNDI

Camel uses a Registry to allow you to configure Component or Endpoint instances or Beans used in your routes. If you are not using Spring or OSGi then JNDI is used as the default registry implementation.

So you will also need to create a `jndi.properties` file in your `src/test/resources` directory so that there is a default registry available to initialize the CamelContext.

Here is an example `jndi.properties` file

```
java.naming.factory.initial = org.apache.camel.util.jndi.CamelInitialContextFactory
```

Dynamically Assigning Ports

Available as of Camel 2.7

Tests that use port numbers will fail if that port is already on use. `AvailablePortFinder` provides methods for finding unused port numbers at run time.

```
java// Get the next available port number starting from the default starting port of 1024 int port1 = AvailablePortFinder.getNextAvailable(); /* Get another port. Note that just getting a port number does not reserve it so * we look starting one past the last port number we got. */ int port2 = AvailablePortFinder.getNextAvailable(port1 + 1);
```

Setup CamelContext once per class, or per every test method

Available as of Camel 2.8

The Camel Test kit will by default setup and shutdown CamelContext per every test method in your test class. So for example if you have 3 test methods, then CamelContext is started and shutdown after each test, that is 3 times.

TestNG

This feature is also supported in camel-testng

Beware

When using this the CamelContext will keep state between tests, so have that in mind. So if your unit tests start to fail for no apparent reason, it could be due to this fact. So use this feature with a bit of care.

You may want to do this once, to share the CamelContext between test methods, to speedup unit testing. This requires the use of JUnit 4! In your unit test method you have to extend the `org.apache.camel.test.junit4.CamelTestSupport` or the `org.apache.camel.test.junit4.CamelSpringTestSupport` test class and override the `isCreateCamelContextPerClass` method and return `true` as shown in the following example:{snippet:id=example|lang=java|title=Setup CamelContext once per class|url=camel/trunk/components/camel-test/src/test/java/org/apache/camel/test/patterns/FilterCreateCamelContextPerClassTest.java}

See Also

- Testing
- Mock
- Test

SPRING TESTING

Testing is a crucial part of any development or integration work. The Spring Framework offers a number of features that makes it easy to test while using Spring for Inversion of Control which works with JUnit 3.x, JUnit 4.x, and TestNG.

We can use Spring for IoC and the Camel Mock and Test endpoints to create sophisticated integration/unit tests that are easy to run and debug inside your IDE. There are three supported approaches for testing with Spring in Camel.

Name	Testing Frameworks Supported	Description
CamelSpringTestSupport	<ul style="list-style-type: none"> • JUnit 3.x (deprecated) • JUnit 4.x • TestNG - Camel 2.8 	<p>Provided by:</p> <ul style="list-style-type: none"> • org.apache.camel.test.CamelSpringTestSupport • org.apache.camel.test.junit4.CamelSpringTestSupport • org.apache.camel.testng.CamelSpringTestSupport <p>These base classes provide feature parity with the simple CamelTestSupport from Camel Test but do not support Spring annotations on the test class such as @Autowired, @DirtiesContext, and @ContextConfiguration.</p>
Plain Spring Test	<ul style="list-style-type: none"> • JUnit 3.x • JUnit 4.x • TestNG 	<p>Either extend the abstract base classes:</p> <ul style="list-style-type: none"> • org.springframework.test.context.junit38.AbstractJUnit38SpringContextRunner • org.springframework.test.context.junit38.AbstractJUnit4SpringContextRunner • etc. <p>provided in Spring Test or use the Spring Test JUnit4 runner.</p> <p>These approaches support both the Camel annotations and Spring annotations. They do NOT have feature parity with:</p> <ul style="list-style-type: none"> • org.apache.camel.test.CamelTestSupport • org.apache.camel.test.junit4.CamelTestSupport • org.apache.camel.testng.CamelSpringTestSupport
Camel Enhanced Spring Test	<ul style="list-style-type: none"> • JUnit 4.x - Camel 2.10 • TestNG - Camel 2.10 	<p>Either:</p> <ul style="list-style-type: none"> • use the org.apache.camel.test.junit4.CamelSpringJUnit4ClassRunner class with the @RunWith annotation, • or extend org.apache.camel.testng.AbstractCamelTestNGSpringContextRunner to enable feature parity with org.apache.camel.test.CamelTestSupport and org.apache.camel.test.junit4.CamelTestSupport. These classes support a suite of Spring Test annotations such as @Autowired, @DirtiesContext, and @ContextConfiguration.

CamelSpringTestSupport

The following Spring test support classes:

- org.apache.camel.test.CamelSpringTestSupport
- org.apache.camel.test.junit4.CamelSpringTestSupport, and
- org.apache.camel.testng.CamelSpringTestSupport

extend their non-Spring aware counterparts:

- org.apache.camel.test.CamelTestSupport
- org.apache.camel.test.junit4.CamelTestSupport, and
- org.apache.camel.testng.CamelTestSupport

and deliver integration with Spring into your test classes.

Instead of instantiating the `CamelContext` and routes programmatically, these classes rely on a Spring context to wire the needed components together. If your test extends one of these classes, you must provide the Spring context by implementing the following method.

```
java@protected abstract ApplicationContext createApplicationContext();
```

You are responsible for the instantiation of the Spring context in the method implementation. All of the features available in the non-Spring aware counterparts from Camel Test are available in your test.

Plain Spring Test

In this approach, your test classes directly inherit from the Spring Test abstract test classes or use the JUnit 4.x test runner provided in Spring Test. This approach supports dependency injection into your test class and the full suite of Spring Test annotations. However, it does not support the features provided by the `CamelSpringTestSupport` classes.

Plain Spring Test using JUnit 3.x with XML Config Example

Here is a simple unit test using JUnit 3.x support from Spring Test using XML Config.

```
{snippet:lang=java|id=example|url=camel/trunk/components/camel-
```

```
spring/src/test/java/org/apache/camel/spring/patterns/FilterTest.java}Notice that we use @DirtiesContext on the test methods to force Spring Testing to automatically reload the CamelContext after each test method - this ensures that the tests don't clash with each other, e.g., one test method sending to an endpoint that is then reused in another test method.
```

Also notice the use of `@ContextConfiguration` to indicate that by default we should look for the file `FilterTest-context.xml` on the classpath to configure the test case. The test context looks like:

```
{snippet:lang=xml|id=example|url=camel/trunk/components/camel-
```

```
spring/src/test/resources/org/apache/camel/spring/patterns/FilterTest-context.xml}This test will load a Spring XML configuration file called FilterTest-context.xml from the classpath in the same package structure as the FilterTest class and initialize it along with any Camel routes we define inside it, then inject the CamelContext instance into our test case.
```

For instance, like this maven folder layout:

```
src/test/java/org/apache/camel/spring/patterns/FilterTest.java src/test/resources/org/apache/camel/spring/patterns/FilterTest-context.xml
```

Plain Spring Test Using JUnit 4.x With Java Config Example

You can completely avoid using an XML configuration file by using Spring Java Config. Here is a unit test using JUnit 4.x support from Spring Test using Java Config.

```
{snippet:lang=java|id=example|url=camel/trunk/components/camel-spring- javaconfig/src/test/java/org/apache/camel/spring/javaconfig/patterns/FilterTest.java}This is similar to the XML Config example above except that there is no XML file and instead the nested ContextConfig class does all of the configuration; so your entire test case is contained in a single Java class. We currently have to reference by class name this class in the @ContextConfiguration which is a bit ugly. Please vote for SJC-238 to address this and make Spring Test work more cleanly with Spring JavaConfig.
```

Plain Spring Test Using JUnit 4.0.x Runner With XML Config

You can avoid extending Spring classes by using the `SpringJUnit4ClassRunner` provided by Spring Test. This custom JUnit runner means you are free to choose your own class hierarchy while retaining all the capabilities of Spring Test.

This is for Spring 4.0.x. If you use Spring 4.1 or newer, then see the next section.

```
java@RunWith(SpringJUnit4ClassRunner.class) @ContextConfiguration public class MyCamelTest { @Autowired protected CamelContext camelContext; @EndpointInject(uri = "mock:foo") protected MockEndpoint foo; @Test @DirtiesContext public void testMocksAreValid() throws Exception { // ... foo.message(0).header("bar").isEqualTo("ABC"); MockEndpoint.assertIsSatisfied(camelContext); } }
```

Plain Spring Test Using JUnit 4.1.x Runner With XML Config

You can avoid extending Spring classes by using the `SpringJUnit4ClassRunner` provided by Spring Test. This custom JUnit runner means you are free to choose your own class hierarchy while retaining all the capabilities of Spring Test.

From **Spring 4.1**, you need to use the `@BootstrapWith` annotation to configure it to use Camel testing, as shown below.

```
java@RunWith(CamelSpringJUnit4ClassRunner.class) @BootstrapWith(CamelTestContextBootstrapper.class) @ContextConfiguration public class MyCamelTest { @Autowired protected CamelContext camelContext; @EndpointInject(uri = "mock:foo") protected MockEndpoint foo; @Test @DirtiesContext public void testMocksAreValid() throws Exception { // ... foo.message(0).header("bar").isEqualTo("ABC"); MockEndpoint.assertIsSatisfied(camelContext); } }
```

Camel Enhanced Spring Test

Using the `org.apache.camel.test.junit4.CamelSpringJUnit4ClassRunner` runner with the `@RunWith` annotation or extending `org.apache.camel.testing.AbstractCamelTestNGSpringContextTests` provides the full feature set of Spring Test with support for the feature set provided in the `CamelTestSupport` classes.

A number of Camel specific annotations have been developed in order to provide for declarative manipulation of the Camel context(s) involved in the test. These annotations free your test classes from having to inherit from the `CamelSpringTestSupport` classes and also reduce the amount of code required to customize the tests.

Annotation Class	Applies To	Description
<code>org.apache.camel.test.spring.disableJmx</code>	class	Indicates if JMX should be bootstrapped during the application contexts.
<code>org.apache.camel.test.spring.ExcludeRoutes</code>	class	Indicates if certain route to be initialized. It initializes a <code>org.apache.camel.test.spring.ExcludeRoutes</code> annotation to exclude a set of given camel routes at test time to exclude certain routes being discovered and initialized.
<code>org.apache.camel.test.spring.LazyLoadTypeConverters</code>	class	Deprecated. Indicates if the CamelContext should be initialized through the use of Spring loading of type converters.
<code>org.apache.camel.test.spring.MockEndpoints</code>	class	Triggers the auto-mocking filter. The default filter is <code>See org.apache.camel.test.spring.MockEndpoints</code> . See <code>org.apache.camel.test.spring.MockEndpoints</code> for more details on the registration.
<code>org.apache.camel.test.spring.MockEndpointsAndSkip</code>	class	Triggers the auto-mocking filter. The default filter is <code>See org.apache.camel.test.spring.MockEndpointsAndSkip</code> . See <code>org.apache.camel.test.spring.MockEndpointsAndSkip</code> for more details on the registration and skip sending the message.
<code>org.apache.camel.test.spring.ProvidesBreakpoint</code>	Method	Indicates that the annotated method intercepts traffic to all endpoints in the CamelContext. This is useful for debugging. The <code>ProvidesBreakpoint</code> annotation can be used in combination with an IDE for debugging. The return value of the method must be <code>org.apache.camel.test.spring.Breakpoint</code> .
<code>org.apache.camel.test.spring.ShutdownTimeout</code>	class	Indicates to set the shutdown timeout. Through the use of Spring beans, the <code>ShutdownTimeout</code> annotation is used, the timeout is automatically applied to the CamelContext.
<code>org.apache.camel.test.spring.UseAdviceWith</code>	class	Indicates the use of advice with. The <code>UseAdviceWith</code> annotation is used to annotated with this annotation to enable advice with. The test author is responsible for writing the advice and executing CamelContext after any advice has been applied.
<code>org.apache.camel.test.spring.UseOverridePropertiesWithPropertiesComponent</code>	Method	Camel 2.16: Indicates that the <code>PropertiesComponent</code> is used to override any existing properties.

The following example illustrates the use of the `@MockEndpoints` annotation in order to setup mock endpoints as interceptors on all endpoints using the Camel Log component and the `@disableJmx` annotation to enable JMX which is disabled during tests by default.

Note: we still use the `@DirtiesContext` annotation to ensure that the CamelContext, routes, and mock endpoints are reinitialized between test methods.

```
java@RunWith(CamelSpringJUnit4ClassRunner.class)
@BootstrapWith(CamelTestContextBootstrapper.class) @ContextConfiguration @DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD) @MockEndpoints("log:*) @DisableJmx(false) public class CamelSpringJUnit4ClassRunnerPlainTest { @Autowired protected CamelContext camelContext2; protected MockEndpoint mockB; @EndpointInject(uri = "mock:c", context = "camelContext2") protected MockEndpoint mockC; @Produce(uri = "direct:start2", context = "camelContext2") protected ProducerTemplate start2; @EndpointInject(uri = "mock:log:org.apache.camel.test.junit4.spring", context = "camelContext2") protected MockEndpoint mockLog; @Test public void testPositive() throws Exception { mockC.expectedBodiesReceived("David"); mockLog.expectedBodiesReceived("Hello David"); start2.sendBody("David"); MockEndpoint.assertIsSatisfied(camelContext); }}
```

Adding More Mock Expectations

```
If you wish to programmatically add any new assertions to your test you can easily do so with the following. Notice how we use @EndpointInject to inject a Camel endpoint into our code then the Mock API to add an expectation on a specific message.
java@ContextConfiguration public class MyCamelTest extends AbstractJUnit38SpringContextTests { @Autowired protected CamelContext camelContext; @EndpointInject(uri = "mock:foo") protected MockEndpoint foo; public void testMocksAreValid() throws Exception { // lets add more expectations foo.message(0).header("bar").isEqualTo("ABC"); MockEndpoint.assertIsSatisfied(camelContext); }}
```

Further Processing the Received Messages

Sometimes once a Mock endpoint has received some messages you want to then process them further to add further assertions that your test case worked as you expect.

```
So you can then process the received message exchanges if you like...
java@ContextConfiguration public class MyCamelTest extends AbstractJUnit38SpringContextTests { @Autowired protected CamelContext camelContext; @EndpointInject(uri = "mock:foo") protected MockEndpoint foo; public void testMocksAreValid() throws Exception { // lets add more expectations... MockEndpoint.assertIsSatisfied(camelContext); // now lets do some further assertions List<Exchange> list = foo.getReceivedExchanges(); for (Exchange exchange : list) { Message in = exchange.getIn(); // ... }}
```

Sending and Receiving Messages

It might be that the Enterprise Integration Patterns you have defined in either Spring XML or using the Java DSL do all of the sending and receiving and you might just work with the Mock endpoints as described above. However sometimes in a test case its useful to explicitly send or receive messages directly.

To send or receive messages you should use the Bean Integration mechanism. For example to send messages inject a ProducerTemplate using the @EndpointInject annotation then call the various send methods on this object to send a message to an endpoint. To consume messages use the @MessageDriven annotation on a method to have the method invoked when a message is received.

```
javapublic class Foo { @EndpointInject(uri = "activemq:foo.bar") ProducerTemplate producer; public void doSomething() { // lets send a message! producer.sendBody("<hello>world!</hello>"); } // lets consume messages from the 'cheese' queue @MessageDriven(uri="activemq:cheese") public void onCheese(String name) { // ... }}
```

See Also

- A real example test case using Mock and Spring along with its Spring XML
- Bean Integration
- Mock endpoint
- Test endpoint

CAMEL GUICE

We have support for Google Guice as a dependency injection framework.

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

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

Dependency Injecting Camel with Guice

The GuiceCamelContext is designed to work nicely inside Guice. You then need to bind it using some Guice Module.

The camel-guice library comes with a number of reusable Guice Modules you can use if you wish - or you can bind the GuiceCamelContext yourself in your own module.

- CamelModule is the base module which binds the GuiceCamelContext but leaves it up to you to bind the RouteBuilder instances
- CamelModuleWithRouteTypes extends CamelModule so that in the constructor of the module you specify the RouteBuilder classes or instances to use
- CamelModuleWithMatchingRoutes extends CamelModule so that all bound RouteBuilder instances will be injected into the CamelContext or you can supply an optional Matcher to find RouteBuilder instances matching some kind of predicate.

So you can specify the exact RouteBuilder instances you want

```
Injector injector = Guice.createInjector(new CamelModuleWithRouteTypes(MyRouteBuilder.class, AnotherRouteBuilder.class));
// if required you can lookup the CamelContext
CamelContext camelContext = injector.getInstance(CamelContext.class);
```

Or inject them all

```
Injector injector = Guice.createInjector(new CamelModuleWithRouteTypes());
// if required you can lookup the CamelContext
CamelContext camelContext = injector.getInstance(CamelContext.class);
```

You can then use Guice in the usual way to inject the route instances or any other dependent objects.

Bootstrapping with JNDI

A common pattern used in J2EE is to bootstrap your application or root objects by looking them up in JNDI. This has long been the approach when working with JMS for example - looking up the JMS ConnectionFactory in JNDI for example.

You can follow a similar pattern with Guice using the GuiceyFruit JNDI Provider which lets you bootstrap Guice from a **jndi.properties** file which can include the Guice Modules to create along with environment specific properties you can inject into your modules and objects.

If the **jndi.properties** is conflict with other component, you can specify the jndi properties file name in the Guice Main with option **-j** or **-jndiProperties** with the properties file location to let Guice Main to load right jndi properties file.

Configuring Component, Endpoint or RouteBuilder instances

You can use Guice to dependency inject whatever objects you need to create, be it an Endpoint, Component, RouteBuilder or arbitrary bean used within a route.

The easiest way to do this is to create your own Guice Module class which extends one of the above module classes and add a provider method for each object you wish to create. A provider method is annotated with **@Provides** as follows

```
public class MyModule extends CamelModuleWithMatchingRoutes {
    @Provides
    @JndiBind("jms")
    JmsComponent jms(@Named("activemq.brokerURL") String brokerUrl) {
        return JmsComponent.jmsComponent(new ActiveMQConnectionFactory(brokerUrl));
    }
}
```

You can optionally annotate the method with **@JndiBind** to bind the object to JNDI at some name if the object is a component, endpoint or bean you wish to refer to by name in your routes.

You can inject any environment specific properties (such as URLs, machine names, usernames/passwords and so forth) from the **jndi.properties** file easily using the **@Named** annotation as shown above. This allows most of your configuration to be in Java code which is typesafe and easily refactorable - then leaving some properties to be environment specific (the **jndi.properties** file) which you can then change based on development, testing, production etc.

Creating multiple RouteBuilder instances per type

It is sometimes useful to create multiple instances of a particular RouteBuilder with different configurations.

To do this just create multiple provider methods for each configuration; or create a single provider method that returns a collection of RouteBuilder instances.

For example

```
import org.apache.camel.guice.CamelModuleWithMatchingRoutes;
import com.google.common.collect.Lists;

public class MyModule extends CamelModuleWithMatchingRoutes {
    @Provides
    @JndiBind("foo")
    Collection<RouteBuilder> foo(@Named("fooUrl") String fooUrl) {
        return Lists.newArrayList(new MyRouteBuilder(fooUrl), new MyRouteBuilder("activemq:CheeseQueue"));
    }
}
```

See Also

- there are a number of Examples you can look at to see Guice and Camel being used such as Guice JMS Example
- Guice Maven Plugin for running your Guice based routes via Maven

TEMPLATING

When you are testing distributed systems its a very common requirement to have to stub out certain external systems with some stub so that you can test other parts of the system until a specific system is available or written etc.

A great way to do this is using some kind of Template system to generate responses to requests generating a dynamic message using a mostly-static body.

There are a number of templating components included in the Camel distribution you could use

- FreeMarker
- Mustache
- StringTemplate
- Velocity
- XQuery
- XSLT

or the following external Camel components

- Scalate

Example

Here's a simple example showing how we can respond to InOut requests on the **My.Queue** queue on ActiveMQ with a template generated response. The reply would be sent back to the JMSReplyTo Destination.

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

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");
```

See Also

- Mock for details of mock endpoint testing (as opposed to template based stubs).

DATABASE

Camel can work with databases in a number of different ways. This document tries to outline the most common approaches.

Database endpoints

Camel provides a number of different endpoints for working with databases

- JPA for working with hibernate, openjpa or toplink. When consuming from the endpoints entity beans are read (and deleted/updated to mark as processed) then when producing to the endpoints they are written to the database (via insert/update).
- iBATIS similar to the above but using Apache iBATIS
- JDBC similar though using explicit SQL
- SQL uses spring-jdbc behind the scene for the actual SQL handling. The difference between this component and JDBC component is that in case of SQL the query is a property of the endpoint and it uses message payload as parameters passed to the query

Database pattern implementations

Various patterns can work with databases as follows

- Idempotent Consumer
- Aggregator
- BAM for business activity monitoring

PARALLEL PROCESSING AND ORDERING

It is a common requirement to want to use parallel processing of messages for throughput and load balancing, while at the same time process certain kinds of messages in order.

How to achieve parallel processing

You can send messages to a number of Camel Components to achieve parallel processing and load balancing such as

- SEDA for in-JVM load balancing across a thread pool
- ActiveMQ or JMS for distributed load balancing and parallel processing
- JPA for using the database as a poor mans message broker

When processing messages concurrently, you should consider ordering and concurrency issues. These are described below

Concurrency issues

Note that there is no concurrency or locking issue when using ActiveMQ, JMS or SEDA by design; they are designed for highly concurrent use. However there are possible concurrency issues in the Processor of the messages i.e. what the processor does with the message?

For example if a processor of a message transfers money from one account to another account; you probably want to use a database with pessimistic locking to ensure that operation takes place atomically.

Ordering issues

As soon as you send multiple messages to different threads or processes you will end up with an unknown ordering across the entire message stream as each thread is going to process messages concurrently.

For many use cases the order of messages is not too important. However for some applications this can be crucial. e.g. if a customer submits a purchase order version 1, then amends it and sends version 2; you don't want to process the first version last (so that you loose the update). Your Processor might be clever enough to ignore old messages. If not you need to preserve order.

Recommendations

This topic is large and diverse with lots of different requirements; but from a high level here are our recommendations on parallel processing, ordering and concurrency

- for distributed locking, use a database by default, they are very good at it 😊
- to preserve ordering across a JMS queue consider using Exclusive Consumers in the ActiveMQ component
- even better are Message Groups which allows you to preserve ordering across messages while still offering parallelisation via the **JMSXGroupID** header to determine what can be parallelized
- if you receive messages out of order you could use the Resequencer to put them back together again

A good rule of thumb to help reduce ordering problems is to make sure each single can be processed as an atomic unit in parallel (either without concurrency issues or using say, database locking); or if it can't, use a Message Group to relate the messages together which need to be processed in order by a single thread.

Using Message Groups with Camel

To use a Message Group with Camel you just need to add a header to the output JMS message based on some kind of Correlation Identifier to correlate messages which should be processed in order by a single thread - so that things which don't correlate together can be processed concurrently.

For example the following code shows how to create a message group using an XPath expression taking an invoice's product code as the Correlation Identifier

```
from("activemq:a").setHeader(JmsConstants.JMS_X_GROUP_ID, xpath("/invoice/productCode")).to("activemq:b");
```

You can of course use the XML Configuration if you prefer

ASYNCHRONOUS PROCESSING

Overview

Supported versions

The information on this page applies for **Camel 2.4** or later.

Before **Camel 2.4** the asynchronous processing is only implemented for JBI where as in **Camel 2.4** we have implemented it in many other areas. See more at Asynchronous Routing Engine.

Camel supports a more complex asynchronous processing model. The asynchronous processors implement the `org.apache.camel.AsyncProcessor` interface which is derived from the more synchronous `org.apache.camel.Processor` interface. There are advantages and disadvantages when using asynchronous processing when compared to using the standard synchronous processing model.

Advantages:

- Processing routes that are composed fully of asynchronous processors do not use up threads waiting for processors to complete on blocking calls. This can increase the scalability of your system by reducing the number of threads needed to process the same workload.
- Processing routes can be broken up into SEDA processing stages where different thread pools can process the different stages. This means that your routes can be processed concurrently.

Disadvantages:

- Implementing asynchronous processors is more complex than implementing the synchronous versions.

When to Use

We recommend that processors and components be implemented the more simple synchronous APIs unless you identify a performance or scalability requirement that dictates otherwise. A Processor whose `process()` method blocks for a long time would be good candidates for being converted into an asynchronous processor.

Interface Details

```
public interface AsyncProcessor extends Processor {
    boolean process(Exchange exchange, AsyncCallback callback);
}
```

The `AsyncProcessor` defines a single `process()` method which is very similar to its synchronous `Processor.process()` brethren.

Here are the differences:

- A non-null `AsyncCallback` **MUST** be supplied which will be notified when the exchange processing is completed.
- It **MUST** not throw any exceptions that occurred while processing the exchange. Any such exceptions must be stored on the exchange's `Exception` property.
- It **MUST** know if it will complete the processing synchronously or asynchronously. The method will return `true` if it does complete synchronously, otherwise it returns `false`.
- When the processor has completed processing the exchange, it must call the `callback.done(boolean sync)` method.
- The `sync` parameter **MUST** match the value returned by the `process()` method.

Implementing Processors that Use the `AsyncProcessor` API

All processors, even synchronous processors that do not implement the `AsyncProcessor` interface, can be coerced to implement the `AsyncProcessor` interface. This is usually done when you are implementing a Camel component consumer that supports asynchronous completion of the exchanges that it is pushing through the Camel routes. Consumers are provided a `Processor` object when created. All `Processor` object can be coerced to a `AsyncProcessor` using the following API:

```
Processor processor = ...
AsyncProcessor asyncProcessor = AsyncProcessorTypeConverter.convert(processor);
```

For a route to be fully asynchronous and reap the benefits to lower Thread usage, it must start with the consumer implementation making use of the asynchronous processing API. If it called the synchronous `process()` method instead, the consumer's thread would be forced to be blocked and in use for the duration that it takes to process the exchange.

It is important to take note that just because you call the asynchronous API, it does not mean that the processing will take place asynchronously. It only allows the possibility that it can be done without tying up the caller's thread. If the processing happens asynchronously is dependent on the configuration of the Camel route.

Normally, the `process` call is passed in an inline inner `AsyncCallback` class instance which can reference the exchange object that was declared final. This allows it to finish up any post processing that is needed when the called processor is done processing the exchange.

Example.

```
final Exchange exchange = ...
AsyncProcessor asyncProcessor = ...
asyncProcessor.process(exchange, new AsyncCallback() {
    public void done(boolean sync) {
        if (exchange.isFailed()) {
            ... // do failure processing.. perhaps rollback etc.
        } else {
            ... // processing completed successfully, finish up
            // perhaps commit etc.
        }
    });
});
```

Asynchronous Route Sequence Scenarios

Now that we have understood the interface contract of the `AsyncProcessor`, and have seen how to make use of it when calling processors, let's look at what the thread model/sequence scenarios will look like for some sample routes.

The Jetty component's consumers support asynchronous processing through the use of continuations. Suffice to say it can take a HTTP request and pass it to a Camel route for asynchronous processing. If the processing is indeed asynchronous, it uses a Jetty continuation so that the HTTP request is 'parked' and the thread is released. Once the Camel route finishes processing the request, the Jetty component uses the `AsyncCallback` to tell Jetty to 'un-park' the request. Jetty un-parks the request, the HTTP response returned using the result of the exchange processing.

Notice that the jetty continuations feature is only used "If the processing is indeed async". This is why `AsyncProcessor.process()` implementations *must* accurately report if request is completed synchronously or not.

The jhc component's producer allows you to make HTTP requests and implement the `AsyncProcessor` interface. A route that uses both the jetty asynchronous consumer and the jhc asynchronous producer will be a fully asynchronous route and has some

nice attributes that can be seen if we take a look at a sequence diagram of the processing route.

For the route:

```
from("jetty:http://localhost:8080/service")
    .to("jhc:http://localhost/service-impl")
```

The sequence diagram would look something like this:



The diagram simplifies things by making it look like processors implement the `AsyncCallback` interface when in reality the `AsyncCallback` interfaces are inline inner classes, but it illustrates the processing flow and shows how two separate threads are used to complete the processing of the original HTTP request. The first thread is synchronous up until processing hits the `jhc` producer which issues the HTTP request. It then reports that the exchange processing will complete asynchronously using NIO to get the response back. Once the `jhc` component has received a full response it uses `AsyncCallback.done()` method to notify the caller. These callback notifications continue up until it reaches the original Jetty consumer which then un-parks the HTTP request and completes it by providing the response.

Mixing Synchronous and Asynchronous Processors

It is totally possible and reasonable to mix the use of synchronous and asynchronous processors/components. The pipeline processor is the backbone of a Camel processing route. It glues all the processing steps together. It is implemented as an `AsyncProcessor` and supports interleaving synchronous and asynchronous processors as the processing steps in the pipeline.

Let's say we have two custom asynchronous processors, namely: `MyValidator` and `MyTransformation`. Let's say we want to load file from the `data/in` directory validate them with the `MyValidator()` processor, transform them into JPA Java objects using `MyTransformation` and then insert them into the database using the `JPA` component. Let's say that the transformation process takes quite a bit of time and we want to allocate 20 threads to do parallel transformations of the input files. The solution is to make use of the thread processor. The thread is `AsyncProcessor` that forces subsequent processing in asynchronous thread from a thread pool.

The route might look like:

```
from("file:data/in")
    .process(new MyValidator())
    .threads(20)
    .process(new MyTransformation())
    .to("jpa:PurchaseOrder");
```

The sequence diagram would look something like this:



You would actually have multiple threads executing the second part of the thread sequence.

Staying Synchronous in an AsyncProcessor

Generally speaking you get better throughput processing when you process things synchronously. This is due to the fact that starting up an asynchronous thread and doing a context switch to it adds a little bit of overhead. So it is generally encouraged that `AsyncProcessor`'s do as much work as they can synchronously. When they get to a step that would block for a long time, at that point they should return from the process call and let the caller know that it will be completing the call asynchronously.

IMPLEMENTING VIRTUAL TOPICS ON OTHER JMS PROVIDERS

ActiveMQ supports Virtual Topics since durable topic subscriptions kinda suck (see this page for more detail) mostly since they don't support Competing Consumers.

Most folks want Queue semantics when consuming messages; so that you can support Competing Consumers for load balancing along with things like Message Groups and Exclusive Consumers to preserve ordering or partition the queue across consumers.

However if you are using another JMS provider you can implement Virtual Topics by switching to ActiveMQ 😊 or you can use the following Camel pattern.

First here's the ActiveMQ approach.

- send to `activemq:topic:VirtualTopic.Orders`
- for consumer A consume from `activemq:Consumer.A.VirtualTopic.Orders`

When using another message broker use the following pattern

- send to `jms:Orders`
- add this route with a `to()` for each logical durable topic subscriber

```
from("jms:Orders").to("jms:Consumer.A", "jms:Consumer.B", ...);
```

- for consumer A consume from **jms:Consumer.A**

WHAT'S THE CAMEL TRANSPORT FOR CXF

In CXF you offer or consume a webservice by defining its address. The first part of the address specifies the protocol to use. For example address="http://localhost:9000" in an endpoint configuration means your service will be offered using the http protocol on port 9000 of localhost. When you integrate Camel Transport into CXF you get a new transport "camel". So you can specify address="camel://direct:MyEndpointName" to bind the CXF service address to a camel direct endpoint.

Technically speaking Camel transport for CXF is a component which implements the CXF transport API with the Camel core library. This allows you to easily use Camel's routing engine and integration patterns support together with your CXF services.

INTEGRATE CAMEL INTO CXF TRANSPORT LAYER

To include the Camel Tranport into your CXF bus you use the CamelTransportFactory. You can do this in Java as well as in Spring.

Setting up the Camel Transport in Spring

You can use the following snippet in your applicationcontext if you want to configure anything special. If you only want to activate the camel transport you do not have to do anything in your application context. As soon as you include the camel-cxf-transport jar (or camel-cxf.jar if your camel version is less than 2.7.x) in your app, cxf will scan the jar and load a CamelTransportFactory for you.

```
xml<!-- you don't need to specify the CamelTransportFactory configuration as it is auto load by CXF bus --> <bean class="org.apache.camel.component.cxf.transport.CamelTransportFactory"> <property name="bus" ref="cxf" /> <property name="camelContext" ref="camelContext" /> <!-- checkException new added in Camel 2.1 and Camel 1.6.2 --> <!-- If checkException is true , CamelDestination will check the outMessage's exception and set it into camel exchange. You can also override this value in CamelDestination's configuration. The default value is false. This option should be set true when you want to leverage the camel's error handler to deal with fault message --> <property name="checkException" value="true" /> <property name="transportIds" /> <list> <value>http://cxf.apache.org/transports/camel</value> </list> </property> </bean>
```

Integrating the Camel Transport in a programmatic way

Camel transport provides a setContext method that you could use to set the Camel context into the transport factory. If you want this factory take effect, you need to register the factory into the CXF bus. Here is a full example for you.

```
javaimport org.apache.cxf.Bus; import org.apache.cxf.BusFactory; import org.apache.cxf.transport.ConduitInitiatorManager; import org.apache.cxf.transport.DestinationFactoryManager; ... BusFactory bf = BusFactory.newInstance(); Bus bus = bf.createBus(); CamelTransportFactory camelTransportFactory = new CamelTransportFactory(); // set up the CamelContext which will be use by the CamelTransportFactory camelTransportFactory.setCamelContext(context) // if you are using CXF higher then 2.4.x the camelTransportFactory.setBus(bus); // if you are lower CXF, you need to register the ConduitInitiatorManager and DestinationFactoryManager like below // register the conduit initiator ConduitInitiatorManager cim = busgetExtension(ConduitInitiatorManager.class); cim.registerConduitInitiator(CamelTransportFactory.TRANSPORT_ID, camelTransportFactory); // register the destination factory DestinationFactoryManager dfm = busgetExtension(DestinationFactoryManager.class); dfm.registerDestinationFactory(CamelTransportFactory.TRANSPORT_ID, camelTransportFactory); // set or bus as the default bus for cxf BusFactory.setDefaultBus(bus);
```

CONFIGURE THE DESTINATION AND CONDUIT WITH SPRING

Namespace

The elements used to configure an Camel transport endpoint are defined in the namespace <http://cxf.apache.org/transports/camel>. It is commonly referred to using the prefix camel1. In order to use the Camel transport configuration elements, you will need to add the lines shown below to the beans element of your endpoint's configuration file. In addition, you will need to add the configuration elements' namespace to the xsi:schemaLocation attribute. Adding the Configuration Namespace<beans ... xmlns:camel="http://cxf.apache.org/transports/camel" ... xsi:schemaLocation="... http://cxf.apache.org/transports/camel http://cxf.apache.org/transports/camel.xsd" ...>

The destination element

You configure an Camel transport server endpoint using the camel1:destination element and its children. The camel1:destination element takes a single attribute, name, that specifies the WSDL port element that corresponds to the endpoint. The value for the name attribute takes the form portQName.camel1-destination. The example below shows the camel1:destination element that would be used to add configuration for an endpoint that was specified by the WSDL fragment <port binding="widgetSOAPBinding" name="widgetSOAPPort"> if the endpoint's target namespace was <http://widgets.widgetvendor.net>.

camel:destination Element... <camel:destination name="<http://widgets/widgetvendor.net>widgetSOAPPort.http-destination> <camelContext id="context" xmlns="http://activemq.apache.org/camel/schema/spring"> <route> <from uri="direct:EndpointC" /> <to uri="direct:EndpointD" /> </route> </camelContext> </camel:destination> <!-- new added feature since Camel 2.11.x --> <camel:destination name="<http://widgets/widgetvendor.net>widgetSOAPPort.camel-destination" camelContextId="context" /> ... The camel:destination element for Spring has a number of child elements that specify configuration information. They are described below.

Element	Description
camel-spring:camelContext	You can specify the camel context in the camel destination
camel:camelContextRef	The camel context id which you want inject into the camel destination

The conduit element

You configure a Camel transport client using the camel:conduit element and its children. The camel:conduit element takes a single attribute, name, that specifies the WSDL port element that corresponds to the endpoint. The value for the name attribute takes the form `portQName.camel-conduit`. For example, the code below shows the camel:conduit element that would be used to add configuration for an endpoint that was specified by the WSDL fragment `<port binding="widgetSOAPBinding" name="widgetSOAPPort" />` if the endpoint's target namespace was <http://widgets.widgetvendor.net>.

```
xmlhttp://conf:conduit Element... <camelContext id="conduit_context" xmlns="http://activemq.apache.org/camel/schema/spring"> <route> <from uri="direct:EndpointA" /> <to uri="direct:EndpointB" /> </route> </camelContext> <camel:conduit name="http://widgets/widgetvendor.netwidgetSOAPPort.camel-conduit"> <camel:camelContextRef>conduit_context</camel:camelContextRef> </camel:conduit> <!-- new added feature since Camel 2.11.x --> <camel:conduit name="http://widgets/widgetvendor.netwidgetSOAPPort.camel-conduit" camelContextId="conduit_context" /> <camel:conduit name="*.camel-conduit"> <!-- you can also using the wild card to specify the camel-conduit that you want to configure --> ... </camel:conduit> ...
```

The camel:conduit element has a number of child elements that specify configuration information. They are described below.

Element	Description
camel-spring:camelContext	You can specify the camel context in the camel conduit
camel:camelContextRef	The camel context id which you want inject into the camel conduit

CONFIGURE THE DESTINATION AND CONDUIT WITH BLUEPRINT

From **Camel 2.11.x**, Camel Transport supports to be configured with Blueprint.

If you are using blueprint, you should use the the namespace <http://cxf.apache.org/transports/camel/blueprint> and import the schema like the blow.

```
Adding the Configuration Namespace for blueprint<beans ... xmlns:camel="http://cxf.apache.org/transports/camel/blueprint" ... xsi:schemaLocation="... http://cxf.apache.org/transports/camel/blueprint http://cxf.apache.org/schemas/blueprint/camel.xsd ...> In blueprint camel:conduit camel:destination only has one camelContextId attribute, they doesn't support to specify the camel context in the camel destination.
<camel:conduit id="*.camel-conduit" camelContextId="camel1" /> <camel:destination id="*.camel-destination" camelContextId="camel1" />
```

EXAMPLE USING CAMEL AS A LOAD BALANCER FOR CXF

This example shows how to use the camel load balancing feature in CXF. You need to load the configuration file in CXF and publish the endpoints on the address "camel://direct:EndpointA" and "camel://direct:EndpointB"

```
{snippet:id=example|lang=xml|url=camel/trunk/examples/camel-example-cxf/src/main/resources/org/apache/camel/example/camel/transport/CamelDestination.xml}
```

COMPLETE HOWTO AND EXAMPLE FOR ATTACHING CAMEL TO CXF

Better JMS Transport for CXF Webservice using Apache Camel

INTRODUCTION

When sending an Exchange to an Endpoint you can either use a Route or a ProducerTemplate. This works fine in many scenarios. However you may need to guarantee that an exchange is delivered to the same endpoint that you delivered a previous exchange on. For example in the case of delivering a batch of exchanges to a MINA socket you may need to ensure that they are all delivered through the same socket connection. Furthermore once the batch of exchanges have been delivered the protocol requirements may be such that you are responsible for closing the socket.

USING A PRODUCER

To achieve fine grained control over sending exchanges you will need to program directly to a Producer. Your code will look similar to:

```
CamelContext camelContext = ...  
  
// Obtain an endpoint and create the producer we will be using.  
Endpoint endpoint = camelContext.getEndpoint("someuri:etc");  
Producer producer = endpoint.createProducer();  
producer.start();  
  
try {  
    // For each message to send...  
    Object requestMessage = ...  
    Exchange exchangeToSend = producer.createExchange();  
    exchangeToSend().setBody(requestMessage);  
    producer.process(exchangeToSend);  
    ...  
}  
finally {  
    // Tidy the producer up.  
    producer.stop();  
}
```

In the case of using Apache MINA the producer.stop() invocation will cause the socket to be closed.

Tutorials

There now follows the documentation on camel tutorials

We have a number of tutorials as listed below. The tutorials often comes with source code which is either available in the Camel Download or attached to the wiki page.

Notice

These tutorials listed below, is hosted at Apache. We offer the Articles page where we have a link collection for 3rd party Camel material, such as tutorials, blog posts, published articles, videos, pod casts, presentations, and so forth.

If you have written a Camel related article, then we are happy to provide a link to it. You can contact the Camel Team, for example using the Mailing Lists, (or post a tweet with the word Apache Camel).

- OAuth Tutorial
This tutorial demonstrates how to implement OAuth for a web application with Camel's gauth component. The sample application of this tutorial is also online at <http://gauthcloud.appspot.com/>
- Tutorial for Camel on Google App Engine
This tutorial demonstrates the usage of the Camel Components for Google App Engine. The sample application of this tutorial is also online at <http://camelcloud.appspot.com/>
- Tutorial on Spring Remoting with JMS
This tutorial is focused on different techniques with Camel for Client-Server communication.
- Report Incident - This tutorial introduces Camel steadily and is based on a real life integration problem
This is a very long tutorial beginning from the start; its for entry level to Camel. Its based on a real life integration, showing how Camel can be introduced in an existing solution. We do this in baby steps. The tutorial is currently work in progress, so check it out from time to time. The tutorial explains some of the inner building blocks Camel uses under the covers. This is good knowledge to have when you start using Camel on a higher abstract level where it can do wonders in a few lines of routing DSL.
- Using Camel with ServiceMix a tutorial on using Camel inside Apache ServiceMix.
- Better JMS Transport for CXF Webservice using Apache Camel Describes how to use the Camel Transport for CXF to attach a CXF Webservice to a JMS Queue
- Tutorial how to use good old Axis 1.4 with Camel
This tutorial shows that Camel does work with the good old frameworks such as AXIS that is/was widely used for WebService.
- Tutorial on using Camel in a Web Application
This tutorial gives an overview of how to use Camel inside Tomcat, Jetty or any other servlet engine
- Tutorial on Camel 1.4 for Integration
Another real-life scenario. The company sells widgets, with a somewhat unique business process (their customers periodically report what they've purchased in order to get billed). However every customer uses a different data format and protocol. This tutorial goes through the process of integrating (and testing!) several customers and their electronic reporting of the widgets they've bought, along with the company's response.
- Tutorial how to build a Service Oriented Architecture using Camel with OSGI - Updated 20/11/2009
The tutorial has been designed in two parts. The first part introduces basic concept to create a simple SOA solution using Camel and OSGI and deploy it in a OSGI Server like Apache Felix Karaf and Spring DM Server while the second extends the ReportIncident tutorial part 4 to show How we can separate the different layers (domain, service, ...) of an application and deploy them in separate bundles. The Web Application has also be modified in order to communicate to the OSGI bundles.
- Several of the vendors on the Commercial Camel Offerings page also offer various tutorials, webinars, examples, etc.... that may be useful.
- Examples
While not actual tutorials you might find working through the source of the various Examples useful.

TUTORIAL ON SPRING REMOTING WITH JMS

Thanks

This tutorial was kindly donated to Apache Camel by Martin Gilday.

PREFACE

This tutorial aims to guide the reader through the stages of creating a project which uses Camel to facilitate the routing of messages from a JMS queue to a Spring service. The route works in a synchronous fashion returning a response to the client.

PREREQUISITES

This tutorial uses Maven to setup the Camel project and for dependencies for artifacts.

DISTRIBUTION

This sample is distributed with the Camel distribution as examples/camel-example-spring-jms.

ABOUT

This tutorial is a simple example that demonstrates more the fact how well Camel is seamless integrated with Spring to leverage the best of both worlds. This sample is client server solution using JMS messaging as the transport. The sample has two flavors of servers and also for clients demonstrating different techniques for easy communication.

The Server is a JMS message broker that routes incoming messages to a business service that does computations on the received message and returns a response.

The EIP patterns used in this sample are:

Pattern	Description
Message Channel	We need a channel so the Clients can communicate with the server.
Message	The information is exchanged using the Camel Message interface.
Message Translator	This is where Camel shines as the message exchange between the Server and the Clients are text based strings with numbers. However our business service uses int for numbers. So Camel can do the message translation automatically.
Message Endpoint	It should be easy to send messages to the Server from the clients. This is achieved with Camel's powerful Endpoint pattern that even can be more powerful combined with Spring remoting. The tutorial has clients using each kind of technique for this.
Point to Point Channel	The client and server exchange data using point to point using a JMS queue.
Event Driven Consumer	The JMS broker is event driven and is invoked when the client sends a message to the server.

We use the following Camel components:

Component	Description
ActiveMQ	We use Apache ActiveMQ as the JMS broker on the Server side
Bean	We use the bean binding to easily route the messages to our business service. This is a very powerful component in Camel.
File	In the AOP enabled Server we store audit trails as files.
JMS	Used for the JMS messaging

CREATE THE CAMEL PROJECT

For the purposes of the tutorial a single Maven project will be used for both the client and server. Ideally you would break your application down into the appropriate components.

```
mvn archetype:generate -DgroupId=org.example -DartifactId=CamelWithJmsAndSpring
```

Update the POM with Dependencies

First we need to have dependencies for the core Camel jars, spring, jms components, and finally ActiveMQ as the message broker.

for the ActiveMQ JMS broker we need this dependency:{snippet:id=e2|lang=xml|url=camel/trunk/examples/camel-example-spring-jms/pom.xml}

WRITING THE SERVER

Create the Spring Service

For this example the Spring service (our business service) on the server will be a simple multiplier which trebles in the received value.{snippet:id=e1|lang=java|url=camel/trunk/examples/camel-example-spring-jms/src/main/java/org/apache/camel/example/server/Multiplier.java}And the implementation of this service is:{snippet:id=e1|lang=java|url=camel/trunk/examples/camel-example-spring-jms/src/main/java/org/apache/camel/example/server/Treble.java}Notice that this class has been annotated with the @Service spring annotation. This ensures that this class is registered as a bean in the registry with the given name **multiplier**.

Define the Camel Routes

{snippet:id=e1|lang=java|url=camel/trunk/examples/camel-example-spring-jms/src/main/java/org/apache/camel/example/server/ServerRoutes.java}This defines a Camel route *from* the JMS queue named **numbers** to the Spring bean named **multiplier**. Camel will create a consumer to the JMS queue which forwards all received messages onto the the Spring bean, using the method named **multiply**.

Configure Spring

The Spring config file is placed under META-INF/spring as this is the default location used by the Camel Maven Plugin, which we will later use to run our server.

First we need to do the standard schema declarations in the top. In the camel-server.xml we are using spring beans as the default **bean**: namespace and springs **context**:. For configuring ActiveMQ we use **broker**: and for Camel we of course have **camel**:. Notice that we don't use version numbers for the camel-spring schema. At runtime the schema is resolved in the Camel bundle. If we use a specific version number such as 1.4 then its IDE friendly as it would be able to import it and provide smart completion etc. See Xml Reference for further details.{snippet:id=e1|lang=xml|url=camel/trunk/examples/camel-example-spring-jms/src/main/resources/META-INF/spring/camel-server.xml}We use Spring annotations for doing IoC dependencies and its component-scan features comes to the rescue as it scans for spring annotations in the given package name:{snippet:id=e2|lang=xml|url=camel/trunk/examples/camel-example-spring-jms/src/main/resources/META-INF/spring/camel-server.xml}Camel will of course not be less than Spring in this regard so it supports a similar feature for scanning of Routes. This is configured as shown below.

Notice that we also have enabled the JMXAgent so we will be able to introspect the Camel Server with a JMX Console.{snippet:id=e3|lang=xml|url=camel/trunk/examples/camel-example-spring-jms/src/main/resources/META-INF/spring/camel-server.xml}The ActiveMQ JMS broker is also configured in this xml file. We set it up to listen on TCP port 61610.{snippet:id=e4|lang=xml|url=camel/trunk/examples/camel-example-spring-jms/src/main/resources/META-INF/spring/camel-server.xml}As this examples uses JMS then Camel needs a JMS component that is connected with the ActiveMQ broker. This is configured as shown below:{snippet:id=e5|lang=xml|url=camel/trunk/examples/camel-example-spring-jms/src/main/resources/META-INF/spring/camel-server.xml}Notice: The JMS component is configured in standard Spring beans, but the gem is that the bean id can be referenced from Camel routes - meaning we can do routing using the JMS Component by just using **jms**: prefix in the route URI. What happens is that Camel will find in the Spring Registry for a bean with the id="jms". Since the bean id can have arbitrary name you could have named it id="jmsbroker" and then referenced to it in the routing as from="jmsbroker:queue:numbers".to("multiplier");

We use the vm protocol to connect to the ActiveMQ server as its embedded in this application.

component-scan	Defines the package to be scanned for Spring stereotype annotations, in this case, to load the "multiplier" bean
camel-context	Defines the package to be scanned for Camel routes. Will find the <code>ServerRoutes</code> class and create the routes contained within it
jms bean	Creates the Camel JMS component

Run the Server

The Server is started using the `org.apache.camel.spring.Main` class that can start camel-spring application out-of-the-box. The Server can be started in several flavors:

- as a standard java main application - just start the `org.apache.camel.spring.Main` class
- using maven `java:exec`
- using `camel:run`

In this sample as there are two servers (with and without AOP) we have prepared some profiles in maven to start the Server of your choice.

The server is started with:

```
mvn compile exec:java -P CamelServer
```

WRITING THE CLIENTS

This sample has three clients demonstrating different Camel techniques for communication

- CamelClient using the ProducerTemplate for Spring template style coding
- CamelRemoting using Spring Remoting
- CamelEndpoint using the Message Endpoint EIP pattern using a neutral Camel API

Client Using The ProducerTemplate

We will initially create a client by directly using ProducerTemplate. We will later create a client which uses Spring remoting to hide the fact that messaging is being used.{snippet:id=e1|lang=xml|url=camel/trunk/examples/camel-example-spring-jms/src/main/resources/camel-client.xml}{snippet:id=e2|lang=xml|url=camel/trunk/examples/camel-example-spring-jms/src/main/resources/camel-client.xml}{snippet:id=e3|lang=xml|url=camel/trunk/examples/camel-example-spring-jms/src/main/resources/camel-client.xml}The client will not use the Camel Maven Plugin so the Spring XML has been placed in `src/main/resources` to not conflict with the server configs.

`camelContext` The Camel context is defined but does not contain any routes

`template` The ProducerTemplate is used to place messages onto the JMS queue

`jms bean` This initialises the Camel JMS component, allowing us to place messages onto the queue

And the CamelClient source code:{snippet:id=e1|lang=java|url=camel/trunk/examples/camel-example-spring-jms/src/main/java/org/apache/camel/example/client/CamelClient.java}The ProducerTemplate is retrieved from a Spring ApplicationContext and used to manually place a message on the "numbers" JMS queue. The `requestBody` method will use the exchange pattern InOut, which states that the call should be synchronous, and that the caller expects a response.

Before running the client be sure that both the ActiveMQ broker and the CamelServer are running.

Client Using Spring Remoting

Spring Remoting "eases the development of remote-enabled services". It does this by allowing you to invoke remote services through your regular Java interface, masking that a remote service is being called.

{snippet:id=e1|lang=xml|url=camel/trunk/examples/camel-example-spring-jms/src/main/resources/camel-client-remoting.xml}The snippet above only illustrates the different and how Camel easily can setup and use Spring Remoting in one line configurations.

The proxy will create a proxy service bean for you to use to make the remote invocations. The `serviceInterface` property details which Java interface is to be implemented by the proxy. The `serviceUrl` defines where messages sent to this proxy bean will be directed. Here we define the JMS endpoint with the "numbers" queue we used when working with Camel template directly. The value of the `id` property is the name that will be given to the bean when it is exposed through the Spring ApplicationContext. We will use this name to retrieve the service in our client. I have named the bean `multiplierProxy` simply to highlight that it is not the same multiplier bean as is being used by CamelServer. They are in completely independent contexts and have no knowledge of each other. As you are trying to mask the fact that remoting is being used in a real application you would generally not include proxy in the name.

And the Java client source code:{snippet:id=e1|lang=java|url=camel/trunk/examples/camel-example-spring-jms/src/main/java/org/apache/camel/example/client/CamelClientRemoting.java}Again, the client is similar to the original client, but with some important differences.

1. The Spring context is created with the new `camel-client-remoting.xml`
2. We retrieve the proxy bean instead of a ProducerTemplate. In a non-trivial example you would have the bean injected as in the standard Spring manner.
3. The multiply method is then called directly. In the client we are now working to an interface. There is no mention of Camel or JMS inside our Java code.

Client Using Message Endpoint EIP Pattern

This client uses the Message Endpoint EIP pattern to hide the complexity to communicate to the Server. The Client uses the same simple API to get hold of the endpoint, create an exchange that holds the message, set the payload and create a producer that does the send and receive. All done using the same neutral Camel API for **all** the components in Camel. So if the communication was socket TCP based you just get hold of a different endpoint and all the java code stays the same. That is really powerful.

Okay enough talk, show me the code!{snippet:id=e1|lang=java|url=camel/trunk/examples/camel-example-spring-jms/src/main/java/org/apache/camel/example/client/CamelClientEndpoint.java}Switching to a different component is just a matter of using the correct endpoint. So if we had defined a TCP endpoint as: "`mina:tcp://localhost:61610`" then its just a matter of getting hold of this endpoint instead of the JMS and all the rest of the java code is exactly the same.

Run the Clients

The Clients is started using their main class respectively.

- as a standard java main application - just start their main class
- using maven `java:exec`

In this sample we start the clients using maven:

```
mvn compile exec:java -PCamelClient
mvn compile exec:java -PCamelClientRemoting
mvn compile exec:java -PCamelClientEndpoint
```

Also see the Maven pom.xml file how the profiles for the clients is defined.

USING THE CAMEL MAVEN PLUGIN

The Camel Maven Plugin allows you to run your Camel routes directly from Maven. This negates the need to create a host application, as we did with Camel server, simply to start up the container. This can be very useful during development to get Camel routes running quickly.
 pom.xml<build> <plugins> <plugin> <groupId>org.apache.camel</groupId> <artifactId>camel-maven-plugin</artifactId> </plugin>
 </plugins> </build>
 All that is required is a new plugin definition in your Maven POM. As we have already placed our Camel config in the default location (camel-server.xml has been placed in META-INF/spring/) we do not need to tell the plugin where the route definitions are located. Simply run mvn camel:run.

USING CAMEL JMX

Camel has extensive support for JMX and allows us to inspect the Camel Server at runtime. As we have enabled the JMXAgent in our tutorial we can fire up the jconsole and connect to the following service URI:
 service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/camel. Notice that Camel will log at INFO level the JMX Connector URI:
 ... DefaultInstrumentationAgent INFO JMX connector thread started on service:jmx:rmi:///jndi/rmi://claus-acer:1099/jmxrmi/camel

In the screenshot below we can see the route and its performance metrics:



SEE ALSO

- Spring Remoting with JMS Example on Amin Abbaspour's Weblog

TUTORIAL - CAMEL-EXAMPLE-REPORTINCIDENT

INTRODUCTION

Creating this tutorial was inspired by a real life use-case I discussed over the phone with a colleague. He was working at a client whom uses a heavy-weight integration platform from a very large vendor. He was in talks with developer shops to implement a new integration on this platform. His trouble was the shop tripled the price when they realized the platform of choice. So I was wondering how we could do this integration with Camel. Can it be done, without tripling the cost 😊.

This tutorial is written during the development of the integration. I have decided to start off with a sample that isn't Camel's but standard Java and then plugin Camel as we goes. Just as when people needed to learn Spring you could consume it piece by piece, the same goes with Camel.

The target reader is person whom hasn't experience or just started using Camel.

MOTIVATION FOR THIS TUTORIAL

I wrote this tutorial motivated as Camel lacked an example application that was based on the web application deployment model. The entire world hasn't moved to pure OSGi deployments yet. The full source code for this tutorial as complete is part of the Apache Camel distribution in the examples/camel-example-reportincident directory

THE USE-CASE

The goal is to allow staff to report incidents into a central administration. For that they use client software where they report the incident and submit it to the central administration. As this is an integration in a transition phase the administration should get these incidents by email whereas they are manually added to the database. The client software should gather the incident and submit the information to the integration platform that in term will transform the report into an email and send it to the central administrator for manual processing.

The figure below illustrates this process. The end users reports the incidents using the client applications. The incident is sent to the central integration platform as webservice. The integration platform will process the incident and send an OK

acknowledgment back to the client. Then the integration will transform the message to an email and send it to the administration mail server. The users in the administration will receive the emails and take it from there.



In EIP patterns

We distill the use case as EIP patterns:



PARTS

This tutorial is divided into sections and parts:

Section A: Existing Solution, how to slowly use Camel

Part 1 - This first part explain how to setup the project and get a webservice exposed using Apache CXF. In fact we don't touch Camel yet.

Part 2 - Now we are ready to introduce Camel piece by piece (without using Spring or any XML configuration file) and create the full feature integration. This part will introduce different Camel's concepts and How we can build our solution using them like :

- CamelContext
- Endpoint, Exchange & Producer
- Components : Log, File

Part 3 - Continued from part 2 where we implement that last part of the solution with the event driven consumer and how to send the email through the Mail component.

Section B: The Camel Solution

Part 4 - We now turn into the path of Camel where it excels - the routing.

Part 5 - Is about how embed Camel with Spring and using CXF endpoints directly in Camel

Part 6 - Showing a alternative solution primarily using XML instead of Java code

Using Axis 2

See this blog entry by Sagara demonstrating how to use Apache Axis 2 instead of Apache CXF as the web service framework.

LINKS

- Introduction
- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6

PART 1

PREREQUISITES

This tutorial uses the following frameworks:

- Maven 3.0.4
- Apache Camel 2.10.0
- Apache CXF 2.6.1
- Spring 3.0.7

Note: The sample project can be downloaded, see the resources section.

INITIAL PROJECT SETUP

We want the integration to be a standard .war application that can be deployed in any web container such as Tomcat, Jetty or even heavy weight application servers such as WebLogic or WebSphere. There fore we start off with the standard Maven webapp project that is created with the following long archetype command:

```
mvn archetype:create -DgroupId=org.apache.camel -DartifactId=camel-example-reportincident -DarchetypeArtifactId=maven-archetype-webapp
```

Notice that the groupId etc. doesn't have to be org.apache.camel it can be com.mycompany.whatever. But I have used these package names as the example is an official part of the Camel distribution.

Then we have the basic maven folder layout. We start out with the webservice part where we want to use Apache CXF for the webservice stuff. So we add this to the pom.xml

```
<properties>
    <cxf-version>2.6.1</cxf-version>
</properties>

<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxfrt-core</artifactId>
    <version>${cxf-version}</version>
</dependency>
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxfrt-frontend-jaxws</artifactId>
    <version>${cxf-version}</version>
</dependency>
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxfrt-transports-http</artifactId>
    <version>${cxf-version}</version>
</dependency>
```

DEVELOPING THE WEBSERVICE

As we want to develop webservice with the contract first approach we create our .wsdl file. As this is a example we have simplified the model of the incident to only include 8 fields. In real life the model would be a bit more complex, but not to much.

We put the wsdl file in the folder `src/main/webapp/WEB-INF/wsdl` and name the file `report_incident.wsdl`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://reportincident.example.camel.apache.org"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://reportincident.example.camel.apache.org">

    <!-- Type definitions for input- and output parameters for webservice -->
    <wsdl:types>
        <xss:schema targetNamespace="http://reportincident.example.camel.apache.org">
            <xss:element name="inputReportIncident">
                <xss:complexType>
                    <xss:sequence>
                        <xss:element type="xs:string" name="incidentId"/>
                        <xss:element type="xs:string" name="incidentDate"/>
                        <xss:element type="xs:string" name="givenName"/>
                        <xss:element type="xs:string" name="familyName"/>
                        <xss:element type="xs:string" name="summary"/>
                        <xss:element type="xs:string" name="details"/>
                        <xss:element type="xs:string" name="email"/>
                        <xss:element type="xs:string" name="phone"/>
                    </xss:sequence>
                </xss:complexType>
            </xss:element>
            <xss:element name="outputReportIncident">
                <xss:complexType>
                    <xss:sequence>
                        <xss:element type="xs:string" name="code"/>
                    </xss:sequence>
                </xss:complexType>
            </xss:element>
        </xss:schema>
    </wsdl:types>

    <!-- Message definitions for input and output -->
    <wsdl:message name="inputReportIncident">
        <wsdl:part name="parameters" element="tns:inputReportIncident"/>
    </wsdl:message>
    <wsdl:message name="outputReportIncident">
        <wsdl:part name="parameters" element="tns:outputReportIncident"/>
    </wsdl:message>

    <!-- Port (interface) definitions -->
    <wsdl:portType name="ReportIncidentEndpoint">
        <wsdl:operation name="ReportIncident">
            <wsdl:input message="tns:inputReportIncident"/>
            <wsdl:output message="tns:outputReportIncident"/>
        </wsdl:operation>
    </wsdl:portType>

    <!-- Port bindings to transports and encoding - HTTP, document literal encoding is used -->
    <wsdl:binding name="ReportIncidentBinding" type="tns:ReportIncidentEndpoint">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http">
            <wsdl:operation name="ReportIncident">
                <soap:operation
                    soapAction="http://reportincident.example.camel.apache.org/ReportIncident"
                    style="document"/>
                <wsdl:input>
                    <soap:body parts="parameters" use="literal"/>
                </wsdl:input>
                <wsdl:output>
                    <soap:body parts="parameters" use="literal"/>
                </wsdl:output>
            </wsdl:operation>
        </wsdl:binding>
    <!-- Service definition -->
    <wsdl:service name="ReportIncidentService">
        <wsdl:port name="ReportIncidentPort" binding="tns:ReportIncidentBinding">
            <soap:address location="http://reportincident.example.camel.apache.org"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

```
</wsdl:service>
</wsdl:definitions>
```

CXF wsdl2java

Then we integration the CXF wsdl2java generator in the pom.xml so we have CXF generate the needed POJO classes for our webservice contract.

However at first we must configure maven to live in the modern world of Java 1.6 so we must add this to the pom.xml

```
<!-- to compile with 1.6 -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.6</source>
    <target>1.6</target>
  </configuration>
</plugin>
```

And then we can add the CXF wsdl2java code generator that will hook into the compile goal so its automatic run all the time:

```
<!-- CXF wsdl2java generator, will plugin to the compile goal -->
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf-version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>${basedir}/target/generated/src/main/java</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl1>${basedir}/src/main/webapp/WEB-INF/wsdl/report_incident.wsdl</wsdl1>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goal>wsdl2java</goal>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

You are now setup and should be able to compile the project. So running the `mvn compile` should run the CXF wsdl2java and generate the source code in the folder `{basedir}/target/generated/src/main/java` that we specified in the pom.xml above. Since its in the `target/generated/src/main/java` maven will pick it up and include it in the build process.

Configuration of the web.xml

Next up is to configure the web.xml to be ready to use CXF so we can expose the webservice.

As Spring is the center of the universe, or at least is a very important framework in today's Java land we start with the listener that kick-starts Spring. This is the usual piece of code:

```
<!-- the listener that kick-starts Spring -->
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

And then we have the CXF part where we define the CXF servlet and its URI mappings to which we have chosen that all our webservices should be in the path /webservices/

```
<!-- CXF servlet -->
<servlet>
  <servlet-name>CXFServlet</servlet-name>
  <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<!-- all our webservices are mapped under this URI pattern -->
<servlet-mapping>
  <servlet-name>CXFServlet</servlet-name>
  <url-pattern>/webservices/*</url-pattern>
</servlet-mapping>
```

Then the last piece of the puzzle is to configure CXF, this is done in a spring XML that we link to from the web.xml by the standard Spring contextConfigLocation property in the web.xml

```
<!-- location of spring xml files -->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:cxf-config.xml</param-value>
</context-param>
```

We have named our CXF configuration file `cxf-config.xml` and its located in the root of the classpath. In Maven land that is we can have the `cxf-config.xml` file in the `src/main/resources` folder. We could also have the file located in the WEB-INF folder for instance `<param-value>/WEB-INF/cxf-config.xml</param-value>`.

Getting rid of the old jsp world

The maven archetype that created the basic folder structure also created a sample .jsp file `index.jsp`. This file `src/main/webapp/index.jsp` should be deleted.

Configuration of CXF

The `cxf-config.xml` is as follows:

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml"/>
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

    <!-- implementation of the webservice -->
    <bean id="reportIncidentEndpoint" class="org.apache.camel.example.reportincident.ReportIncidentEndpointImpl"/>

    <!-- export the webservice using jaxws -->
    <jaxws:endpoint id="reportIncident"
        implementor="#reportIncidentEndpoint"
        address="/incident"
        wsdlLocation="WEB-INF/wsdl/report_incident.wsdl"
        endpointName="s:ReportIncidentPort"
        serviceName="s:ReportIncidentService"
        xmlns:s="http://reportincident.example.camel.apache.org"/>

</beans>

```

The configuration is standard CXF and is documented at the Apache CXF website.

The 3 import elements is needed by CXF and they must be in the file.

Noticed that we have a spring bean **reportIncidentEndpoint** that is the implementation of the webservice endpoint we let CXF expose.

Its linked from the jaxws element with the implementor attribute as we use the # mark to identify its a reference to a spring bean. We could have stated the classname directly as

`implementor="org.apache.camel.example.reportincident.ReportIncident"` but then we lose the ability to let the ReportIncidentEndpoint be configured by spring.

The **address** attribute defines the relative part of the URL of the exposed webservice. **wsdlLocation** is an optional parameter but for persons like me that likes contract-first we want to expose our own .wsdl contracts and not the auto generated by the frameworks, so with this attribute we can link to the real .wsdl file. The last stuff is needed by CXF as you could have several services so it needs to know which this one is. Configuring these is quite easy as all the information is in the wsdl already.

Implementing the ReportIncidentEndpoint

Phew after all these meta files its time for some java code so we should code the implementor of the webservice. So we fire up `mvn compile` to let CXF generate the POJO classes for our webservice and we are ready to fire up a Java editor.

You can use `mvn idea:idea` or `mvn eclipse:eclipse` to create project files for these editors so you can load the project. However IDEA has been smarter lately and can load a pom.xml directly.

As we want to quickly see our webservice we implement just a quick and dirty as it can get. At first beware that since its jaxws and Java 1.5 we get annotations for the money, but they reside on the interface so we can remove them from our implementations so its a nice plain POJO again:

```

package org.apache.camel.example.reportincident;

/**
 * The webservice we have implemented.
 */
public class ReportIncidentEndpointImpl implements ReportIncidentEndpoint {

    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        System.out.println("Hello ReportIncidentEndpointImpl is called from " + parameters.getGivenName());

        OutputReportIncident out = new OutputReportIncident();
        out.setCode("OK");
        return out;
    }
}

```

We just output the person that invokes this webservice and returns a OK response. This class should be in the maven source root folder `src/main/java` under the package name `org.apache.camel.example.reportincident`. Beware that the maven archetype tool didn't create the `src/main/java` folder, so you should create it manually.

To test if we are home free we run `mvn clean compile`.

Running our webservice

Now that the code compiles we would like to run it inside a web container, for this purpose we make use of Jetty which we will bootstrap using its plugin `org.mortbay.jetty:maven-jetty-plugin`:

```
<build>
  <plugins>
    <!-- so we can run mvn jetty:run -->
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty-plugin</artifactId>
      <version>${jetty-version}</version>
    </plugin>
```

Notice: We make use of the Jetty version being defined inside the Camel's Parent POM.

So to see if everything is in order we fire up jetty with `mvn jetty:run` and if everything is okay you should be able to access `http://localhost:8080`.

Jetty is smart that it will list the correct URI on the page to our web application, so just click on the link. This is smart as you don't have to remember the exact web context URI for your application - just fire up the default page and Jetty will help you.

So where is the damn webservice then? Well as we did configure the web.xml to instruct the CXF servlet to accept the pattern `/webservices/*` we should hit this URL to get the attention of CXF: `http://localhost:8080/camel-example-reportincident/webservices`.



Hitting the webservice

Now we have the webservice running in a standard .war application in a standard web container such as Jetty we would like to invoke the webservice and see if we get our code executed. Unfortunately this isn't the easiest task in the world - its not so easy as a REST URL, so we need tools for this. So we fire up our trusty webservice tool SoapUI and let it be the one to fire the webservice request and see the response.

Using SoapUI we sent a request to our webservice and we got the expected OK response and the console outputs the System.out so we are ready to code.



Remote Debugging

Okay a little sidestep but wouldn't it be cool to be able to debug your code when its fired up under Jetty? As Jetty is started from maven, we need to instruct maven to use debug mode.

Se we set the MAVEN_OPTS environment to start in debug mode and listen on port 5005.

```
MAVEN_OPTS=-Xmx512m -XX:MaxPermSize=128m -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005
```

Then you need to restart Jetty so its stopped with `ctrl + c`. Remember to start a new shell to pickup the new environment settings. And start jetty again.

Then we can from our IDE attach a remote debugger and debug as we want. First we configure IDEA to attach to a remote debugger on port 5005:



Then we set a breakpoint in our code `ReportIncidentEndpoint` and hit the SoapUI once again and we are breaked at the breakpoint where we can inspect the parameters:



Adding a unit test

Oh so much hard work just to hit a webservice, why can't we just use an unit test to invoke our webservice? Yes of course we can do this, and that's the next step.

First we create the folder structure `src/test/java` and `src/test/resources`. We then create the unit test in the `src/test/java` folder.

```
package org.apache.camel.example.reportincident;
import junit.framework.TestCase;

/**
 * Plain JUnit test of our webservice.
 */
public class ReportIncidentEndpointTest extends TestCase {
```

Here we have a plain old JUnit class. As we want to test webservices we need to start and expose our webservice in the unit test before we can test it. And JAXWS has pretty decent methods to help us here, the code is simple as:

```
import javax.xml.ws.Endpoint;
...
private static String ADDRESS = "http://localhost:9090/unittest";
protected void startServer() throws Exception {
    // We need to start a server that exposes or webservice during the unit testing
    // We use jaxws to do this pretty simple
    ReportIncidentEndpointImpl server = new ReportIncidentEndpointImpl();
    Endpoint.publish(ADDRESS, server);
}
```

The Endpoint class is the `javax.xml.ws.Endpoint` that under the covers looks for a provider and in our case its CXF - so its CXF that does the heavy lifting of exposing out webservice on the given URL address. Since our class `ReportIncidentEndpointImpl` implements the interface `ReportIncidentEndpoint` that is decorated with all the jaxws annotations it got all the information it need to expose the webservice. Below is the CXF wsdl2java generated interface:

```
/*
 */
package org.apache.camel.example.reportincident;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.ParameterStyle;
import javax.xml.bind.annotation.XmlSeeAlso;

/*
 * This class was generated by Apache CXF 2.1.1
 * Wed Jul 16 12:40:31 CEST 2008
 * Generated source version: 2.1.1
 */
/*
 */
/*
 */
/*
 */

@WebService(targetNamespace = "http://reportincident.example.camel.apache.org", name = "ReportIncidentEndpoint")
@XmlSeeAlso({ObjectFactory.class})
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public interface ReportIncidentEndpoint {

/*
 */
/*
 */

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "outputReportIncident", targetNamespace = "http://reportincident.example.camel.apache.org", partName = "parameters")
    @WebMethod(operationName = "ReportIncident", action = "http://reportincident.example.camel.apache.org/ReportIncident")
    public OutputReportIncident reportIncident(
        @WebParam(partName = "parameters", name = "inputReportIncident", targetNamespace = "http://reportincident.example.camel.apache.org")
        InputReportIncident parameters
    );
}
```

Next up is to create a webservice client so we can invoke our webservice. For this we actually use the CXF framework directly as its a bit more easier to create a client using this framework than using the JAXWS style. We could have done the same for the server part, and you should do this if you need more power and access more advanced features.

```
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
...
protected ReportIncidentEndpoint createCXFClient() {
    // We use CXF to create a client for us as its easier than JAXWS and works
    JaxWsProxyFactoryBean<ReportIncidentEndpoint> factory = new JaxWsProxyFactoryBean();
    factory.setServiceClass(ReportIncidentEndpoint.class);
    factory.setAddress(ADDRESS);
    return (ReportIncidentEndpoint) factory.create();
}
```

So now we are ready for creating a unit test. We have the server and the client. So we just create a plain simple unit test method as the usual junit style:

```
public void testReportIncident() throws Exception {
    startServer();

    ReportIncidentEndpoint client = createCXFClient();

    InputReportIncident input = new InputReportIncident();
    input.setIncidentId("123");
    input.setIncidentDate("2008-07-16");
    input.setGivenName("Claus");
    input.setFamilyName("Ibsen");
    input.setSummary("bla bla");
    input.setDetails("more bla bla");
    input.setEmail("davsclaus@apache.org");
```

```

        input.setPhone("+45 2962 7576");
        OutputReportIncident out = client.reportIncident(input);
        assertEquals("Response code is wrong", "OK", out.getCode());
    }
}

```

Now we are nearly there. But if you run the unit test with `mvn test` then it will fail. Why!!! Well its because that CXF needs is missing some dependencies during unit testing. In fact it needs the web container, so we need to add this to our `pom.xml`.

```

<!-- cxf web container for unit testing -->
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http-jetty</artifactId>
    <version>${cxf-version}</version>
    <scope>test</scope>
</dependency>

```

Well what is that, CXF also uses Jetty for unit test - well its just shows how agile, embedable and popular Jetty is.

So lets run our junit test with, and it reports:

```

mvn test
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] BUILD SUCCESSFUL

```

Yep thats it for now. We have a basic project setup.

END OF PART 1

Thanks for being patient and reading all this more or less standard Maven, Spring, JAXWS and Apache CXF stuff. Its stuff that is well covered on the net, but I wanted a full fledged tutorial on a maven project setup that is web service ready with Apache CXF. We will use this as a base for the next part where we demonstrate how Camel can be digested slowly and piece by piece just as it was back in the times when was introduced and was learning the Spring framework that we take for granted today.

#RESOURCES

- Apache CXF user guide

LINKS

- Introduction
- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6

PART 2

ADDING CAMEL

In this part we will introduce Camel so we start by adding Camel to our `pom.xml`:

```

<properties>
    ...
    <camel-version>1.4.0</camel-version>
</properties>

<!-- camel -->
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-core</artifactId>
    <version>${camel-version}</version>
</dependency>

```

That's it, only **one** dependency for now.

Synchronize IDE

If you continue from part 1, remember to update your editor project settings since we have introduce new .jar files. For instance IDEA has a feature to synchronize with Maven projects.

Now we turn towards our webservice endpoint implementation where we want to let Camel have a go at the input we receive. As Camel is very non invasive its basically a .jar file then we can just grap Camel but creating a new instance of `DefaultCamelContext` that is the heart of Camel its context.

```
CamelContext camel = new DefaultCamelContext();
```

In fact we create a constructor in our webservice and add this code:

```
private CamelContext camel;
public ReportIncidentEndpointImpl() throws Exception {
    // create the camel context that is the "heart" of Camel
    camel = new DefaultCamelContext();

    // add the log component
    camel.addComponent("log", new LogComponent());

    // start Camel
    camel.start();
}
```

LOGGING THE "HELLO WORLD"

Here at first we want Camel to log the **givenName** and **familyName** parameters we receive, so we add the LogComponent with the key **log**. And we must **start** Camel before its ready to act.

Component Documentation

The Log and File components is documented as well, just click on the links. Just return to this documentation later when you must use these components for real.

Then we change the code in the method that is invoked by Apache CXF when a webservice request arrives. We get the name and let Camel have a go at it in the new method we create **sendToCamel**:

```
public OutputReportIncident reportIncident(InputReportIncident parameters) {
    String name = parameters.getGivenName() + " " + parameters.getFamilyName();

    // let Camel do something with the name
    sendToCamelLog(name);

    OutputReportIncident out = new OutputReportIncident();
    out.setCode("OK");
    return out;
}
```

Next is the Camel code. At first it looks like there are many code lines to do a simple task of logging the name - yes it is. But later you will in fact realize this is one of Camels true power. Its concise API. Hint: The same code can be used for **any** component in Camel.

```
private void sendToCamelLog(String name) {
    try {
        // get the log component
        Component component = camel.getComponent("log");

        // create an endpoint and configure it.
        // Notice the URI parameters this is a common practice in Camel to configure
        // endpoints based on URI.
        // com.mycompany.part2 the log category used. Will log at INFO level as default
        Endpoint endpoint = component.createEndpoint("log:com.mycompany.part2");

        // create an Exchange that we want to send to the endpoint
        Exchange exchange = endpoint.createExchange();
        // set the in message payload (=body) with the name parameter
        exchange.getIn().setBody(name);

        // now we want to send the exchange to this endpoint and we then need a producer
        // for this, so we create and start the producer.
        Producer producer = endpoint.createProducer();
        producer.start();
        // process the exchange will send the exchange to the log component, that will process
        // the exchange and yes log the payload
        producer.process(exchange);

        // stop the producer, we want to be nice and cleanup
        producer.stop();
    }

    } catch (Exception e) {
        // we ignore any exceptions and just rethrow as runtime
        throw new RuntimeException(e);
    }
}
```

Okay there are code comments in the code block above that should explain what is happening. We run the code by invoking our unit test with maven mvn test, and we should get this log line:

```
INFO: Exchange[BodyType:String, Body:Claus Ibsen]
```

WRITE TO FILE - EASY WITH THE SAME CODE STYLE

Okay that isn't to impressive, Camel can log 😊 Well I promised that the above code style can be used for **any** component, so let's store the payload in a file. We do this by adding the file component to the Camel context

```
// add the file component
camel.addComponent("file", new FileComponent());
```

And then we let camel write the payload to the file after we have logged, by creating a new method **sendToCamelFile**. We want to store the payload in filename with the incident id so we need this parameter also:

```
// let Camel do something with the name
sendToCamelLog(name);
sendToCamelFile(parameters.getIncidentId(), name);
```

And then the code that is 99% identical. We have change the URL configuration when we create the endpoint as we pass in configuration parameters to the file component.

And then we need to set the output filename and this is done by adding a special header to the exchange. That's the only difference:

```
private void sendToCamelFile(String incidentId, String name) {
    try {
        // get the file component
        Component component = camel.getComponent("file");

        // create an endpoint and configure it.
        // Notice the URI parameters this is a common practice in Camel to configure
        // endpoints based on URI.
        // file://target instructs the base folder to output the files. We put in the target folder
        // then its automatically cleaned by mvn clean
        Endpoint endpoint = component.createEndpoint("file://target");

        // create an Exchange that we want to send to the endpoint
        Exchange exchange = endpoint.createExchange();
        // set the in message payload (<body>) with the name parameter
        exchange.getIn().setBody(name);

        // now a special header is set to instruct the file component what the output filename
        // should be
        exchange.getIn().setHeader(FileComponent.HEADER_FILE_NAME, "incident-" + incidentId + ".txt");

        // now we want to send the exchange to this endpoint and we then need a producer
        // for this, so we create and start the producer.
        Producer producer = endpoint.createProducer();
        producer.start();
        // process the exchange will send the exchange to the file component, that will process
        // the exchange and yes write the payload to the given filename
        producer.process(exchange);

        // stop the producer, we want to be nice and cleanup
        producer.stop();
    } catch (Exception e) {
        // we ignore any exceptions and just rethrow as runtime
        throw new RuntimeException(e);
    }
}
```

After running our unit test again with **mvn test** we have a output file in the target folder:

```
D:\demo\part-two>type target\incident-123.txt
Claus Ibsen
```

FULLY JAVA BASED CONFIGURATION OF ENDPOINTS

In the file example above the configuration was URI based. What if you want 100% java setter based style, well this is of course also possible. We just need to cast to the component specific endpoint and then we have all the setters available:

```
// create the file endpoint, we cast to FileEndpoint because then we can do
// 100% java setter based configuration instead of the URI string based
// must pass in an empty string, or part of the URL configuration if wanted
FileEndpoint endpoint = (FileEndpoint)component.createEndpoint("");
endpoint.setFile("new_file(\"target/subfolder\")");
endpoint.setAutoCreate(true);
```

That's it. Now we have used the setters to configure the **FileEndpoint** that it should store the file in the folder target/subfolder. Of course Camel now stores the file in the subfolder.

```
D:\demo\part-two>type target\subfolder\incident-123.txt
Claus Ibsen
```

LESSONS LEARNED

Okay I wanted to demonstrate how you can be in 100% control of the configuration and usage of Camel based on plain Java code with no hidden magic or special XML or other configuration files. Just add the camel-core.jar and you are ready to go.

You must have noticed that the code for sending a message to a given endpoint is the same for both the **log** and **file**, in fact **any** Camel endpoint. You as the client shouldn't bother with component specific code such as file stuff for file components, jms stuff for JMS messaging etc. This is what the Message Endpoint EIP pattern is all about and Camel solves this very very nice - a key pattern in Camel.

REDUCING CODE LINES

Now that you have been introduced to Camel and one of its masterpiece patterns solved elegantly with the Message Endpoint its time to give productive and show a solution in fewer code lines, in fact we can get it down to 5, 4, 3, 2 ... yes only **1 line of code**.

The key is the **ProducerTemplate** that is a Spring'ish `xxxTemplate` based producer. Meaning that it has methods to send messages to any Camel endpoints. First of all we need to get hold of such a template and this is done from the `CamelContext`

```
private ProducerTemplate template;
public ReportIncidentEndpointImpl() throws Exception {
    ...
    // get the ProducerTemplate that is a Spring'ish xxxTemplate based producer for very
    // easy sending exchanges to Camel.
    template = camel.createProducerTemplate();
    ...
    // start Camel
    camel.start();
}
```

Now we can use `template` for sending payloads to any endpoint in Camel. So all the logging gabble can be reduced to:

```
template.sendBody("log:com.mycompany.part2.easy", name);
```

And the same goes for the file, but we must also send the header to instruct what the output filename should be:

```
String filename = "easy-incident-" + incidentId + ".txt";
template.sendBodyAndHeader("file://target/subfolder", name, FileComponent.HEADER_FILE_NAME, filename);
```

REDUCING EVEN MORE CODE LINES

Well we got the Camel code down to 1-2 lines for sending the message to the component that does all the heavy work of wring the message to a file etc. But we still got 5 lines to initialize Camel.

```
camel = new DefaultCamelContext();
camel.addComponent("log", new LogComponent());
camel.addComponent("file", new FileComponent());
template = camel.createProducerTemplate();
camel.start();
```

This can also be reduced. All the standard components in Camel is auto discovered on-the-fly so we can remove these code lines and we are down to 3 lines.

Component auto discovery

When an endpoint is requested with a scheme that Camel hasn't seen before it will try to look for it in the classpath. It will do so by looking for special Camel component marker files that reside in the folder `META-INF/services/org/apache/camel/component`. If there are files in this folder it will read them as the filename is the **scheme** part of the URL. For instance the **log** component is defined in this file `META-INF/services/org/apache/component/log` and its content is:

```
class=org.apache.camel.component.log.LogComponent
```

The class property defines the component implementation.

Tip: End-users can create their 3rd party components using the same technique and have them been auto discovered on-the-fly.

Okay back to the 3 code lines:

```
camel = new DefaultCamelContext();
template = camel.createProducerTemplate();
camel.start();
```

Later will we see how we can reduce this to ... in fact 0 java code lines. But the 3 lines will do for now.

MESSAGE TRANSLATION

Okay lets head back to the over goal of the integration. Looking at the EIP diagrams at the introduction page we need to be able to translate the incoming webservice to an email. Doing so we need to create the email body. When doing the message translation we could put up our sleeves and do it manually in pure java with a `StringBuilder` such as:

```
private String createMailBody(InputReportIncident parameters) {
    StringBuilder sb = new StringBuilder();
    sb.append("Incident ").append(parameters.getIncidentId());
    sb.append(" has been reported on the ").append(parameters.getIncidentDate());
    sb.append(" by ").append(parameters.getGivenName());
    sb.append(" ").append(parameters.getFamilyName());
    // and the rest of the mail body with more appends to the string builder
    return sb.toString();
}
```

But as always it is a hardcoded template for the mail body and the code gets kinda ugly if the mail message has to be a bit more advanced. But of course it just works out-of-the-box with just classes already in the JDK.

Lets use a template language instead such as Apache Velocity. As Camel have a component for Velocity integration we will use this component. Looking at the Component List overview we can see that camel-velocity component uses the artifactId **camel-velocity** so therefore we need to add this to the **pom.xml**:

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

And now we have a Spring conflict as Apache CXF is dependent on Spring 2.0.8 and camel-velocity is dependent on Spring 2.5.5. To remedy this we could wrestle with the **pom.xml** with excludes settings in the dependencies or just bring in another dependency **camel-spring**:

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

In fact **camel-spring** is such a vital part of Camel that you will end up using it in nearly all situations - we will look into how well Camel is seamless integration with Spring in part 3. For now its just another dependency.

We create the mail body with the Velocity template and create the file **src/main/resources/MailBody.vm**. The content in the **MailBody.vm** file is:

```
Incident $body.incidentId has been reported on the $body.incidentDate by $body.givenName $body.familyName.

The person can be contact by:
- email: $body.email
- phone: $body.phone

Summary: $body.summary

Details:
$body.details

This is an auto generated email. You can not reply.
```

Letting Camel creating the mail body and storing it as a file is as easy as the following 3 code lines:

```
private void generateEmailBodyAndStoreAsFile(InputReportIncident parameters) {
    // generate the mail body using velocity template
    // notice that we just pass in our POJO (= InputReportIncident) that we
    // got from Apache CXF to Velocity.
    Object response = template.sendBody("velocity:Mailbody.vm", parameters);
    // Note: the response is a String and can be cast to String if needed

    // store the mail in a file
    String filename = "mail-incident-" + parameters.getIncidentId() + ".txt";
    template.sendBodyAndHeader("file://target/subfolder", response, FileComponent.HEADER_FILE_NAME, filename);
}
```

What is impressive is that we can just pass in our POJO object we got from Apache CXF to Velocity and it will be able to generate the mail body with this object in its context. Thus we don't need to prepare **anything** before we let Velocity loose and generate our mail body. Notice that the **template** method returns a object with out response. This object contains the mail body as a String object. We can cast to String if needed.

If we run our unit test with **mvn test** we can in fact see that Camel has produced the file and we can type its content:

```
D:\demo\part-two>type target\subfolder\mail-incident-123.txt
Incident 123 has been reported on the 2008-07-16 by Claus Ibsen.

The person can be contact by:
- email: davsclaus@apache.org
- phone: +45 2962 7576

Summary: bla bla

Details:
more bla bla

This is an auto generated email. You can not reply.
```

FIRST PART OF THE SOLUTION

What we have seen here is actually what it takes to build the first part of the integration flow. Receiving a request from a webservice, transform it to a mail body and store it to a file, and return an OK response to the webservice. All possible within 10 lines of code. So lets wrap it up here is what it takes:

```
/*
 * The webservice we have implemented.
 */
public class ReportIncidentEndpointImpl implements ReportIncidentEndpoint {

    private CamelContext camel;
    private ProducerTemplate template;

    public ReportIncidentEndpointImpl() throws Exception {
        // create the camel context that is the "heart" of Camel
        camel = new DefaultCamelContext();

        // get the ProducerTemplate thst is a Spring'ish xxxTemplate based producer for very
        // easy sending exchanges to Camel.
        template = camel.createProducerTemplate();

        // start Camel
        camel.start();
    }

    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        // transform the request into a mail body
        Object mailBody = template.sendBody("velocity:MailBody.vm", parameters);

        // store the mail body in a file
        String filename = "mail-incident-" + parameters.getIncidentId() + ".txt";
        template.sendBodyAndHeader("file://target/subfolder", mailBody, FileComponent.HEADER_FILE_NAME, filename);

        // return an OK reply
        OutputReportIncident out = new OutputReportIncident();
        out.setCode("OK");
        return out;
    }
}
```

Okay I missed by one, its in fact only 9 lines of java code and 2 fields.

END OF PART 2

I know this is a bit different introduction to Camel to how you can start using it in your projects just as a plain java .jar framework that isn't invasive at all. I took you through the coding parts that requires 6 - 10 lines to send a message to an endpoint, but it's important to show the Message Endpoint EIP pattern in action and how its implemented in Camel. Yes of course Camel also has to one liners that you can use, and will use in your projects for sending messages to endpoints. This part has been about good old plain java, nothing fancy with Spring, XML files, auto discovery, OGSI or other new technologies. I wanted to demonstrate the basic building blocks in Camel and how its setup in pure god old fashioned Java. There are plenty of eye catcher examples with one liners that does more than you can imagine - we will come there in the later parts.

Okay part 3 is about building the last pieces of the solution and now it gets interesting since we have to wrestle with the event driven consumer.

Brew a cup of coffee, tug the kids and kiss the wife, for now we will have us some fun with the Camel. See you in part 3.

LINKS

- [Introduction](#)
- [Part 1](#)
- [Part 2](#)
- [Part 3](#)
- [Part 4](#)
- [Part 5](#)
- [Part 6](#)

PART 3

RECAP

Lets just recap on the solution we have now:

```
public class ReportIncidentEndpointImpl implements ReportIncidentEndpoint {

    private CamelContext camel;
    private ProducerTemplate template;

    public ReportIncidentEndpointImpl() throws Exception {
        // create the camel context that is the "heart" of Camel
    }
}
```

```

camel = new DefaultCamelContext();
// get the ProducerTemplate that is a Spring'ish xxxTemplate based producer for very
// easy sending exchanges to Camel.
template = camel.createProducerTemplate();
// start Camel
camel.start();
}

/**
 * This is the last solution displayed that is the most simple
 */
public OutputReportIncident reportIncident(InputReportIncident parameters) {
    // transform the request into a mail body
    Object mailBody = template.sendBody("velocity:MailBody.vm", parameters);

    // store the mail body in a file
    String filename = "mail-incident-" + parameters.getIncidentId() + ".txt";
    template.sendBodyAndHeader("file://target/subfolder", mailBody, FileComponent.HEADER_FILE_NAME, filename);

    // return an OK reply
    OutputReportIncident out = new OutputReportIncident();
    out.setCode("OK");
    return out;
}
}

```

This completes the first part of the solution: receiving the message using webservice, transform it to a mail body and store it as a text file.

What is missing is the last part that polls the text files and send them as emails. Here is where some fun starts, as this requires usage of the Event Driven Consumer EIP pattern to react when new files arrives. So lets see how we can do this in Camel. There is a saying: Many roads lead to Rome, and that is also true for Camel - there are many ways to do it in Camel.

ADDING THE EVENT DRIVEN CONSUMER

We want to add the consumer to our integration that listen for new files, we do this by creating a private method where the consumer code lives. We must register our consumer in Camel before its started so we need to add, and there fore we call the method **addMailSenderConsumer** in the constructor below:

```

public ReportIncidentEndpointImpl() throws Exception {
    // create the camel context that is the "heart" of Camel
    camel = new DefaultCamelContext();

    // get the ProducerTemplate that is a Spring'ish xxxTemplate based producer for very
    // easy sending exchanges to Camel.
    template = camel.createProducerTemplate();

    // add the event driven consumer that will listen for mail files and process them
    addMailSendConsumer();

    // start Camel
    camel.start();
}

```

The consumer needs to be consuming from an endpoint so we grab the endpoint from Camel we want to consume. It's `file://target/subfolder`. Don't be fooled this endpoint doesn't have to 100% identical to the producer, i.e. the endpoint we used in the previous part to create and store the files. We could change the URL to include some options, and to make it more clear that it's possible we setup a delay value to 10 seconds, and the first poll starts after 2 seconds. This is done by adding `?consumer.delay=10000&consumer.initialDelay=2000` to the URL.

URL Configuration

The URL configuration in Camel endpoints is just like regular URL we know from the Internet. You use `?` and `&` to set the options. When we have the endpoint we can create the consumer (just as in part 1 where we created a producer). Creating the consumer requires a Processor where we implement the java code what should happen when a message arrives. To get the mail body as a String object we can use the **getBody** method where we can provide the type we want in return.

Camel Type Converter

Why don't we just cast it as we always do in Java? Well the biggest advantage when you provide the type as a parameter you tell Camel what type you want and Camel can automatically convert it for you, using its flexible Type Converter mechanism. This is a great advantage, and you should try to use this instead of regular type casting.

Sending the email is still left to be implemented, we will do this later. And finally we must remember to start the consumer otherwise its not active and won't listen for new files.

```

private void addMailSendConsumer() throws Exception {
    // Grab the endpoint where we should consume. Option - the first poll starts after 2 seconds
    Endpoint endpoint = camel.getEndpoint("file://target/subfolder?consumer.initialDelay=2000");

    // create the event driven consumer
    // the Processor is the code what should happen when there is an event
    // (think it as the onMessage method)
    Consumer consumer = endpoint.createConsumer(new Processor() {
        public void process(Exchange exchange) throws Exception {
            // get the mail body as a String
            String mailBody = exchange.getIn().getBody(String.class);

            // okay now we are ready to send it as an email
            System.out.println("Sending email..." + mailBody);
        }
    });
}

```

```
// start the consumer, it will listen for files
consumer.start();
}
```

Before we test it we need to be aware that our unit test is only catering for the first part of the solution, receiving the message with webservice, transforming it using Velocity and then storing it as a file - it doesn't test the Event Driven Consumer we just added. As we are eager to see it in action, we just do a common trick adding some sleep in our unit test, that gives our Event Driven Consumer time to react and print to System.out. We will later refine the test:

```
public void testRendportIncident() throws Exception {
    ...
    OutputReportIncident out = client.reportIncident(input);
    assertEquals("Response code is wrong", "OK", out.getCode());
    // give the event driven consumer time to react
    Thread.sleep(10 * 1000);
}
```

We run the test with mvn clean test and have eyes fixed on the console output.

During all the output in the console, we see that our consumer has been triggered, as we want.

```
2008-07-19 12:09:24,140 [mponent@1f12c4e] DEBUG FileProcessStrategySupport - Locking the file: target\subfolder\mail-incident-123.txt ...
Sending email...Incident 123 has been reported on the 2008-07-16 by Claus Ibsen.

The person can be contact by:
- email: davsclaus@apache.org
- phone: +45 2962 7576

Summary: bla bla
Details:
more bla bla

This is an auto generated email. You can not reply.
2008-07-19 12:09:24,156 [mponent@1f12c4e] DEBUG FileConsumer - Done processing file: target\subfolder\mail-incident-123.txt. Status is: OK
```

SENDING THE EMAIL

Sending the email requires access to a SMTP mail server, but the implementation code is very simple:

```
private void sendEmail(String body) {
    // send the email to your mail server
    String url = "smtp://someone@localhost?password=secret&to=incident@mycompany.com";
    template.sendBodyAndHeader(url, body, "subject", "New incident reported");
}
```

And just invoke the method from our consumer:

```
// okay now we are ready to send it as an email
System.out.println("Sending email...");
sendEmail(mailBody);
System.out.println("Email sent");
```

UNIT TESTING MAIL

For unit testing the consumer part we will use a mock mail framework, so we add this to our pom.xml:

```
<!-- unit testing mail using mock -->
<dependency>
    <groupId>org.jvnet.mock-javamail</groupId>
    <artifactId>mock-javamail</artifactId>
    <version>1.7</version>
    <scope>test</scope>
</dependency>
```

Then we prepare our integration to run with or without the consumer enabled. We do this to separate the route into the two parts:

- receive the webservice, transform and save mail file and return OK as response
- the consumer that listen for mail files and send them as emails

So we change the constructor code a bit:

```
public ReportIncidentEndpointImpl() throws Exception {
    init(true);
}

public ReportIncidentEndpointImpl(boolean enableConsumer) throws Exception {
    init(enableConsumer);
}

private void init(boolean enableConsumer) throws Exception {
    // create the camel context that is the "heart" of Camel
```

```

camel = new DefaultCamelContext();

// get the ProducerTemplate that is a Spring-ish xxxTemplate based producer for very
// easy sending exchanges to Camel.
template = camel.createProducerTemplate();

// add the event driven consumer that will listen for mail files and process them
if (enableConsumer) {
    addMailSendConsumer();
}

// start Camel
camel.start();
}

```

Then remember to change the `ReportIncidentEndpointTest` to pass in `false` in the `ReportIncidentEndpointImpl` constructor. And as always run `mvn clean test` to be sure that the latest code changes works.

ADDING NEW UNIT TEST

We are now ready to add a new unit test that tests the consumer part so we create a new test class that has the following code structure:

```

/*
 * Plain JUnit test of our consumer.
 */
public class ReportIncidentConsumerTest extends TestCase {

    private ReportIncidentEndpointImpl endpoint;

    public void testConsumer() throws Exception {
        // we run this unit test with the consumer, hence the true parameter
        endpoint = new ReportIncidentEndpointImpl(true);
    }
}


```

As we want to test the consumer that it can listen for files, read the file content and send it as an email to our mailbox we will test it by asserting that we receive 1 mail in our mailbox and that the mail is the one we expect. To do so we need to grab the mailbox with the mockmail API. This is done as simple as:

```

public void testConsumer() throws Exception {
    // we run this unit test with the consumer, hence the true parameter
    endpoint = new ReportIncidentEndpointImpl(true);

    // get the mailbox
    Mailbox box = Mailbox.get("incident@mycompany.com");
    assertEquals("Should not have mails", 0, box.size());
}


```

How do we trigger the consumer? Well by creating a file in the folder it listen for. So we could use plain `java.io.File` API to create the file, but wait isn't there an smarter solution? ... yes Camel of course. Camel can do amazing stuff in one liner codes with its `ProducerTemplate`, so we need to get a hold of this baby. We expose this template in our `ReportIncidentEndpointImpl` but adding this getter:

```

protected ProducerTemplate getTemplate() {
    return template;
}


```

Then we can use the template to create the file in **one code line**:

```

// drop a file in the folder that the consumer listen
// here is a trick to reuse Camel! so we get the producer template and just
// file a message that will create the file for us
endpoint.getTemplate().sendBodyAndHeader("file://target/subfolder?append=false", "Hello World",
    FileComponent.HEADER_FILE_NAME, "mail-incident-test.txt");


```

Then we just need to wait a little for the consumer to kick in and do its work and then we should assert that we got the new mail. Easy as just:

```

// let the consumer have time to run
Thread.sleep(3 * 1000);

// get the mock mailbox and check if we got mail ;
assertEquals("Should have got 1 mail", 1, box.size());
assertEquals("Subject wrong", "New incident reported", box.get(0).getSubject());
assertEquals("Mail body wrong", "Hello World", box.get(0).getContent());
}


```

The final class for the unit test is:

```

/*
 * Plain JUnit test of our consumer.
 */
public class ReportIncidentConsumerTest extends TestCase {

    private ReportIncidentEndpointImpl endpoint;

    public void testConsumer() throws Exception {


```

```
// we run this unit test with the consumer, hence the true parameter
endpoint = new ReportIncidentEndpointImpl(true);

// get the mailbox
Mailbox box = Mailbox.get("incident@mycompany.com");
assertEquals("Should not have mails", 0, box.size());

// drop a file in the folder that the consumer listen
// here is a trick to reuse Camel! so we get the producer template and just
// fire a message that will create the file for us
endpoint.getTemplate().sendBodyAndHeader("file://target/subfolder?append=false", "Hello World",
    FileComponent.HEADER_FILE_NAME, "mail-incident-test.txt");

// let the consumer have time to run
Thread.sleep(3 * 1000);

// get the mock mailbox and check if we got mail ;
assertThat("Should have got 1 mail", 1, box.size());
assertEquals("Subject wrong", "New incident reported", box.get(0).getSubject());
assertEquals("Mail body wrong", "Hello World", box.get(0).getContent());
}

}
```

END OF PART 3

Okay we have reached the end of part 3. For now we have only scratched the surface of what Camel is and what it can do. We have introduced Camel into our integration piece by piece and slowly added more and more along the way. And the most important is: **you as the developer never lost control**. We hit a sweet spot in the webservice implementation where we could write our java code. Adding Camel to the mix is just to use it as a regular java code, nothing magic. We were in control of the flow, we decided when it was time to translate the input to a mail body, we decided when the content should be written to a file. This is very important to not lose control, that the bigger and heavier frameworks tend to do. No names mentioned, but boy do developers from time to time dislike these elephants. And Camel is **no elephant**.

I suggest you download the samples from part 1 to 3 and try them out. It is great basic knowledge to have in mind when we look at some of the features where Camel really excel - **the routing domain language**.

From part 1 to 3 we touched concepts such as::

- Endpoint
- URI configuration
- Consumer
- Producer
- Event Driven Consumer
- Component
- CamelContext
- ProducerTemplate
- Processor
- Type Converter

LINKS

- Introduction
- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6

PART 4

INTRODUCTION

This section is about regular Camel. The examples presented here in this section is much more in common of all the examples we have in the Camel documentation.

If you have been reading the previous 3 parts then, this quote applies:

you must unlearn what you have learned
Master Yoda, Star Wars IV

So we start all over again! 😊

ROUTING

Camel is particularly strong as a light-weight and agile **routing** and **mediation** framework. In this part we will introduce the **routing** concept and how we can introduce this into our solution.

Looking back at the figure from the Introduction page we want to implement this routing. Camel has support for expressing this routing logic using Java as a DSL (Domain Specific Language). In fact Camel also has DSL for XML and Scala. In this part we use the Java DSL as its the most powerful and all developers know Java. Later we will introduce the XML version that is very well integrated with Spring.

Before we jump into it, we want to state that this tutorial is about **Developers not losing control**. In my humble experience one of the key fears of developers is that they are forced into a tool/framework where they lose control and/or power, and the possible is now impossible. So in this part we stay clear with this vision and our starting point is as follows:

- We have generated the webservice source code using the CXF wsdl2java generator and we have our ReportIncidentEndpointImpl.java file where we as a Developer feels home and have the power.

So the starting point is:

```
/*
 * The webservice we have implemented.
 */
public class ReportIncidentEndpointImpl implements ReportIncidentEndpoint {

    /**
     * This is the last solution displayed that is the most simple
     */
    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        // WE ARE HERE !!!
        return null;
    }
}
```

Yes we have a simple plain Java class where we have the implementation of the webservice. The cursor is blinking at the WE ARE HERE block and this is where we feel home. More or less any Java Developers have implemented webservices using a stack such as: Apache AXIS, Apache CXF or some other quite popular framework. They all allow the developer to be in control and implement the code logic as plain Java code. Camel of course doesn't enforce this to be any different. Okay the boss told us to implement the solution from the figure in the Introduction page and we are now ready to code.

RouteBuilder

RouteBuilder is the heart in Camel of the Java DSL routing. This class does all the heavy lifting of supporting EIP verbs for end-users to express the routing. It does take a little while to get settled and used to, but when you have worked with it for a while you will enjoy its power and realize it is in fact a little language inside Java itself. Camel is the **only** integration framework we are aware of that has Java DSL, all the others are usually **only** XML based.

As an end-user you usually use the **RouteBuilder** as of follows:

- create your own Route class that extends **RouteBuilder**
- implement your routing DSL in the **configure** method

So we create a new class ReportIncidentRoutes and implement the first part of the routing:

```
import org.apache.camel.builder.RouteBuilder;

public class ReportIncidentRoutes extends RouteBuilder {

    public void configure() throws Exception {
        // direct:start is an internal queue to kick-start the routing in our example
        // we use this as the starting point where you can send messages to direct:start
        from("direct:start")
            // to is the destination we send the message to our velocity endpoint
            // where we transform the mail body
            .to("velocity:MailBody.vm");
    }
}
```

What to notice here is the **configure** method. Here is where all the action is. Here we have the Java DSL language, that is expressed using the **fluent builder syntax** that is also known from Hibernate when you build the dynamic queries etc. What you do is that you can stack methods separating with the dot.

In the example above we have a very common routing, that can be distilled from pseudo verbs to actual code with:

- from A to B
- From Endpoint A To Endpoint B
- from("endpointA").to("endpointB")
- from("direct:start").to("velocity:MailBody.vm");

from("direct:start") is the consumer that is kick-starting our routing flow. It will wait for messages to arrive on the direct queue and then dispatch the message.

to("velocity:MailBody.vm") is the producer that will receive a message and let Velocity generate the mail body response.

So what we have implemented so far with our ReportIncidentRoutes RouteBuilder is this part of the picture:



Adding the RouteBuilder

Now we have our RouteBuilder we need to add/connect it to our CamelContext that is the heart of Camel. So turning back to our webservice implementation class ReportIncidentEndpointImpl we add this constructor to the code, to create the CamelContext and add the routes from our route builder and finally to start it.

```
private CamelContext context;
public ReportIncidentEndpointImpl() throws Exception {
    // create the context
    context = new DefaultCamelContext();
    // append the routes to the context
    context.addRoutes(new ReportIncidentRoutes());
    // at the end start the camel context
    context.start();
}
```

Okay how do you use the routes then? Well its just as before we use a ProducerTemplate to send messages to Endpoints, so we just send to the `direct:start` endpoint and it will take it from there.

So we implement the logic in our webservice operation:

```
/** 
 * This is the last solution displayed that is the most simple
 */
public OutputReportIncident reportIncident(InputReportIncident parameters) {
    Object mailBody = context.createProducerTemplate().sendBody("direct:start", parameters);
    System.out.println("Body:" + mailBody);

    // return an OK reply
    OutputReportIncident out = new OutputReportIncident();
    out.setCode("OK");
    return out;
}
```

Notice that we get the producer template using the `createProducerTemplate` method on the CamelContext. Then we send the input parameters to the `direct:start` endpoint and it will route it to the velocity endpoint that will generate the mail body. Since we use `direct` as the consumer endpoint (=from) and its a **synchronous** exchange we will get the response back from the route. And the response is of course the output from the velocity endpoint.

About creating ProducerTemplate

In the example above we create a new ProducerTemplate when the `reportIncident` method is invoked. However in reality you should only create the template once and re-use it. See this FAQ entry.

We have now completed this part of the picture:



UNIT TESTING

Now is the time we would like to unit test what we got now. So we call for camel and its great test kit. For this to work we need to add it to the pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-core</artifactId>
    <version>1.4.0</version>
    <scope>test</scope>
    <type>test-jar</type>
</dependency>
```

After adding it to the pom.xml you should refresh your Java Editor so it picks up the new jar. Then we are ready to create our unit test class.

We create this unit test skeleton, where we **extend** this class `ContextTestSupport`

```
package org.apache.camel.example.reportincident;

import org.apache.camel.ContextTestSupport;
import org.apache.camel.builder.RouteBuilder;

/**
 * Unit test of our routes
 */
public class ReportIncidentRoutesTest extends ContextTestSupport {
```

`ContextTestSupport` is a supporting unit test class for much easier unit testing with Apache Camel. The class is extending JUnit `TestCase` itself so you get all its glory. What we need to do now is to somehow tell this unit test class that it should use our route builder as this is the one we gonna test. So we do this by implementing the `createRouteBuilder` method.

```

@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new ReportIncidentRoutes();
}

```

That is easy just return an instance of our route builder and this unit test will use our routes.

It is quite common in Camel itself to unit test using routes defined as an anonymous inner class, such as illustrated below:

```

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // TODO: Add your routes here, such as:
            from("jms:queue:inbox").to("file://target/out");
        }
    };
}

```

The same technique is of course also possible for end-users of Camel to create parts of your routes and test them separately in many test classes.

However in this tutorial we test the real route that is to be used for production, so we just return an instance of the real one.

We then code our unit test method that sends a message to the route and assert that its transformed to the mail body using the Velocity template.

```

public void testTransformMailBody() throws Exception {
    // create a dummy input with some input data
    InputReportIncident parameters = createInput();

    // send the message (using the sendBody method that takes a parameters as the input body)
    // to "direct:start" that kick-starts the route
    // the response is returned as the out object, and its also the body of the response
    Object out = context.createProducerTemplate().sendBody("direct:start", parameters);

    // convert the response to a string using camel converters. However we could also have casted it to
    // a string directly but using the type converters ensure that Camel can convert it if it wasn't a string
    // in the first place. The type converters in Camel is really powerful and you will later learn to
    // appreciate them and wonder why its not build in Java out-of-the-box
    String body = context.getTypeConverter().convertTo(String.class, out);

    // do some simple assertions of the mail body
    assertTrue(body.startsWith("Incident 123 has been reported on the 2008-07-16 by Claus Ibsen."));
}

/**
 * Creates a dummy request to be used for input
 */
protected InputReportIncident createInput() {
    InputReportIncident input = new InputReportIncident();
    input.setIncidentId("123");
    input.setIncidentDate("2008-07-16");
    input.setGivenName("Claus");
    input.setFamilyName("Ibsen");
    input.setSummary("bla bla");
    input.setDetails("more bla bla");
    input.setEmail("davsclaus@apache.org");
    input.setPhone("+45 2962 7576");
    return input;
}

```

ADDING THE FILE BACKUP

The next piece of puzzle that is missing is to store the mail body as a backup file. So we turn back to our route and the EIP patterns. We use the Pipes and Filters pattern here to chain the routing as:

```

public void configure() throws Exception {
    from("direct:start")
        .to("velocity:MailBody.vm")
        // using pipes-and-filters we send the output from the previous to the next
        .to("file://target/subfolder");
}

```

Notice that we just add a 2nd `.to` on the newline. Camel will default use the Pipes and Filters pattern here when there are multi endpoints chained like this. We could have used the `pipeline` verb to let out stand out that its the Pipes and Filters pattern such as:

```

from("direct:start")
    // using pipes-and-filters we send the output from the previous to the next
    .pipeline("velocity:MailBody.vm", "file://target/subfolder");

```

But most people are using the multi `.to` style instead.

We re-run our unit test and verifies that it still passes:

```

Running org.apache.camel.example.reportincident.ReportIncidentRoutesTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.157 sec

```

But hey we have added the `file producer` endpoint and thus a file should also be created as the backup file. If we look in the `target/subfolder` we can see that something happened.

On my humble laptop it created this folder: `target\subfolder\ID-claus-acer`. So the file producer create a sub folder named `ID-`

claus-acer what is this? Well Camel auto generates an unique filename based on the unique message id if not given instructions to use a fixed filename. In fact it creates another sub folder and name the file as: target\subfolderID-claus-acer\3750-1219148558921\1-0 where 1-0 is the file with the mail body. What we want is to use our own filename instead of this auto generated filename. This is archived by adding a header to the message with the filename to use. So we need to add this to our route and compute the filename based on the message content.

Setting the filename

For starters we show the simple solution and build from there. We start by setting a constant filename, just to verify that we are on the right path, to instruct the file producer what filename to use. The file producer uses a special header `FileComponent.HEADER_FILE_NAME` to set the filename.

What we do is to send the header when we "kick-start" the routing as the header will be propagated from the direct queue to the file producer. What we need to do is to use the `ProducerTemplate.sendBodyAndHeader` method that takes **both** a body and a header. So we change out webservice code to include the filename also:

```
public OutputReportIncident reportIncident(InputReportIncident parameters) {
    // create the producer template to use for sending messages
    ProducerTemplate producer = context.createProducerTemplate();
    // send the body and the filename defined with the special header key
    Object mailBody = producer.sendBodyAndHeader("direct:start", parameters, FileComponent.HEADER_FILE_NAME, "incident.txt");
    System.out.println("Body:" + mailBody);

    // return an OK reply
    OutputReportIncident out = new OutputReportIncident();
    out.setCode("OK");
    return out;
}
```

However we could also have used the route builder itself to configure the constant filename as shown below:

```
public void configure() throws Exception {
    from("direct:start")
        .to("velocity:Mailbody.vm")
        // set the filename to a constant before the file producer receives the message
        .setHeader(FileComponent.HEADER_FILE_NAME, constant("incident.txt"))
        .to("file://target/subfolder");
}
```

But Camel can be smarter and we want to dynamic set the filename based on some of the input parameters, how can we do this? Well the obvious solution is to compute and set the filename from the webservice implementation, but then the webservice implementation has such logic and we want this decoupled, so we could create our own POJO bean that has a method to compute the filename. We could then instruct the routing to invoke this method to get the computed filename. This is a string feature in Camel, its Bean binding. So lets show how this can be done:

Using Bean Language to compute the filename

First we create our plain java class that computes the filename, and it has 100% no dependencies to Camel what so ever.

```
/** 
 * Plain java class to be used for filename generation based on the reported incident
 */
public class FilenameGenerator {

    public String generateFilename(InputReportIncident input) {
        // compute the filename
        return "incident-" + input.getIncidentId() + ".txt";
    }
}
```

The class is very simple and we could easily create unit tests for it to verify that it works as expected. So what we want now is to let Camel invoke this class and its `generateFilename` with the input parameters and use the output as the filename. Pheeww is this really possible out-of-the-box in Camel? Yes it is. So lets get on with the show. We have the code that computes the filename, we just need to call it from our route using the Bean Language:

```
public void configure() throws Exception {
    from("direct:start")
        // set the filename using the bean language and call the FilenameGenerator class.
        // the 2nd null parameter is optional methodname, to be used to avoid ambiguity.
        // if not provided Camel will try to figure out the best method to invoke, as we
        // only have one method this is very simple
        .setHeader(FileComponent.HEADER_FILE_NAME, BeanLanguage.bean(FilenameGenerator.class, null))
        .to("velocity:Mailbody.vm")
        .to("file://target/subfolder");
}
```

Notice that we use the **bean** language where we supply the class with our bean to invoke. Camel will instantiate an instance of the class and invoke the suited method. For completeness and ease of code readability we add the method name as the 2nd parameter

```
.setHeader(FileComponent.HEADER_FILE_NAME, BeanLanguage.bean(FilenameGenerator.class, "generateFilename"))
```

Then other developers can understand what the parameter is, instead of null.

Now we have a nice solution, but as a sidetrack I want to demonstrate the Camel has other languages out-of-the-box, and that scripting language is a first class citizen in Camel where it etc. can be used in content based routing. However we want it to be used for the filename generation.

Using a script language to set the filename

We could do as in the previous parts where we send the computed filename as a message header when we "kick-start" the route. But we want to learn new stuff so we look for a different solution using some of Camels many Languages. As OGNL is a favorite language of mine (used by WebWork) so we pick this baby for a Camel ride. For starters we must add it to our pom.xml:

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

And remember to refresh your editor so you got the new jars.

We want to construct the filename based on this syntax: mail-incident-#ID#.txt where #ID# is the incident id from the input parameters. As OGNL is a language that can invoke methods on bean we can invoke the `getIncidentId()` on the message body and then concat it with the fixed pre and postfix strings.

In OGNL glory this is done as:

```
"${mail-incident-} + ${request.body.incidentId} + .txt"
```

where `request.body.incidentId` computes to:

- `request` is the IN message. See the OGNL for other predefined objects available
 - `body` is the body of the in message
 - `incidentId` will invoke the `getIncidentId()` method on the body.
- The rest is just more or less regular plain code where we can concat strings.

Now we got the expression to dynamic compute the filename on the fly we need to set it on our route so we turn back to our route, where we can add the OGNL expression:

```
public void configure() throws Exception {
    from("direct:start")
        // we need to set the filename and uses OGNL for this
        .setHeader(FileComponent.HEADER_FILE_NAME, OgnlExpression.ognl("${mail-incident-} + ${request.body.incidentId} + .txt"));
        // using pipes-and-filters we send the output from the previous to the next
        .pipeline("velocity:MailBody.vm", "file://target/subfolder");
}
```

And since we are on Java 1.5 we can use the static import of `ognl` so we have:

```
import static org.apache.camel.language.ognl.OgnlExpression.ognl;
...
.setHeader(FileComponent.HEADER_FILE_NAME, ognl("${mail-incident-} + ${request.body.incidentId} + .txt"))
```

Notice the import static also applies for all the other languages, such as the Bean Language we used previously. Whatever worked for you we have now implemented the backup of the data files:



SENDING THE EMAIL

What we need to do before the solution is completed is to actually send the email with the mail body we generated and stored as a file. In the previous part we did this with a File consumer, that we manually added to the CamelContext. We can do this quite easily with the routing.

```
import org.apache.camel.builder.RouteBuilder;
public class ReportIncidentRoutes extends RouteBuilder {
    public void configure() throws Exception {
        // first part from the webservice -> file backup
        from("direct:start")
            .setHeader(FileComponent.HEADER_FILE_NAME, bean(FilenameGenerator.class, "generateFilename"))
            .to("velocity:MailBody.vm")
            .to("file://target/subfolder");

        // second part from the file backup -> send email
        from("file://target/subfolder")
            // set the subject of the email
            .setHeader("subject", constant("new incident reported"))
            // send the email
            .to("smtp://someone@localhost?password=secret&to=incident@mycompany.com");
    }
}
```

The last 3 lines of code does all this. It adds a file consumer `from("file://target/subfolder")`, sets the mail subject, and finally send it as an email.

The DSL is really powerful where you can express your routing integration logic.
So we completed the last piece in the picture puzzle with just 3 lines of code.

We have now completed the integration:



CONCLUSION

We have just briefly touched the **routing** in Camel and shown how to implement them using the **fluent builder** syntax in Java. There is much more to the routing in Camel than shown here, but we are learning step by step. We continue in part 5. See you there.

LINKS

- Introduction
- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6

BETTER JMS TRANSPORT FOR CXF WEBSERVICE USING APACHE CAMEL

Configuring JMS in Apache CXF before Version 2.1.3 is possible but not really easy or nice. This article shows how to use Apache Camel to provide a better JMS Transport for CXF.

Update: Since CXF 2.1.3 there is a new way of configuring JMS (Using the JMSConfigFeature). It makes JMS config for CXF as easy as with Camel. Using Camel for JMS is still a good idea if you want to use the rich feature of Camel for routing and other Integration Scenarios that CXF does not support.

You can find the original announcement for this Tutorial and some additional info on Christian Schneider's Blog

So how to connect Apache Camel and CXF

The best way to connect Camel and CXF is using the Camel transport for CXF. This is a camel module that registers with cxf as a new transport. It is quite easy to configure.

```
<bean class="org.apache.camel.component.cxf.transport.CamelTransportFactory">
<property name="bus" ref="cxf" />
<property name="camelContext" ref="camelContext" />
<property name="transportIds">
<list>
<value>http://cxf.apache.org/transports/camel</value>
</list>
</property>
</bean>
```

This bean registers with CXF and provides a new transport prefix `camel://` that can be used in CXF address configurations. The bean references a bean `cxf` which will be already present in your config. The other reference is a camel context. We will later define this bean to provide the routing config.

How is JMS configured in Camel

In camel you need two things to configure JMS. A ConnectionFactory and a JMSComponent. As ConnectionFactory you can simply set up the normal Factory your JMS provider offers or bind a JNDI ConnectionFactory. In this example we use the ConnectionFactory provided by ActiveMQ.

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

Then we set up the JMSComponent. It offers a new transport prefix to camel that we simply call `jms`. If we need several JMSComponents we can differentiate them by their name.

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
<property name="connectionFactory" ref="jmsConnectionFactory" />
<property name="useMessageIDAsCorrelationID" value="true" />
</bean>
```

You can find more details about the JMSComponent at the Camel Wiki. For example you find the complete configuration options and a JNDI sample there.

Setting up the CXF client

We will configure a simple CXF webservice client. It will use stub code generated from a wsdl. The webservice client will be configured to use JMS directly. You can also use a direct: Endpoint and do the routing to JMS in the Camel Context.

```
<client id="CustomerService" xmlns="http://cxf.apache.org/jaxws" xmlns:customer="http://customerservice.example.com/">
    serviceName="customer:CustomerServiceService"
    endpointName="customer:CustomerServiceEndpoint"
    address="camel:jms:queue:CustomerService"
    serviceClass="com.example.customerservice.CustomerService">
</client>
```

We explicitly configure serviceName and endpointName so they are not read from the wsdl. The names we use are arbitrary and have no further function but we set them to look nice. The serviceclass points to the service interface that was generated from the wsdl. Now the important thing is address. Here we tell cxf to use the camel transport, use the JmsComponent who registered the prefix "jms" and use the queue "CustomerService".

Setting up the CamelContext

As we do not need additional routing an empty CamelContext bean will suffice.

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

Running the Example

- Download the example project here
- Follow the readme.txt

Conclusion

As you have seen in this example you can use Camel to connect services to JMS easily while being able to also use the rich integration features of Apache Camel.

TUTORIAL USING AXIS 1.4 WITH APACHE CAMEL

Removed from distribution

This example has been removed from **Camel 2.9** onwards. Apache Axis 1.4 is a very old and unsupported framework. We encourage users to use CXF instead of Axis.

- Tutorial using Axis 1.4 with Apache Camel
 - Prerequisites
 - Distribution
 - Introduction
 - Setting up the project to run Axis
 - Maven 2
 - wsdl
 - Configuring Axis
 - Running the Example
 - Integrating Spring
 - Using Spring
 - Integrating Camel
 - CamelContext
 - Store a file backup
 - Running the example
 - Unit Testing
 - Smarter Unit Testing with Spring
 - Unit Test calling WebService
 - Annotations
 - The End
 - See Also

Prerequisites

This tutorial uses Maven 2 to setup the Camel project and for dependencies for artifacts.

Distribution

This sample is distributed with the Camel 1.5 distribution as examples/camel-example-axis.

Introduction

Apache Axis is/was widely used as a webservice framework. So in line with some of the other tutorials to demonstrate how Camel is not an invasive framework but is flexible and integrates well with existing solution.

We have an existing solution that exposes a webservice using Axis 1.4 deployed as web applications. This is a common solution. We use contract first so we have Axis generated source code from an existing wsdl file. Then we show how we introduce Spring and Camel to integrate with Axis.

This tutorial uses the following frameworks:

- Maven 2.0.9
- Apache Camel 1.5.0
- Apache Axis 1.4
- Spring 2.5.5

Setting up the project to run Axis

This first part is about getting the project up to speed with Axis. We are not touching Camel or Spring at this time.

Maven 2

Axis dependencies is available for maven 2 so we configure our pom.xml as:

```
<dependency>
    <groupId>org.apache.axis</groupId>
    <artifactId>axis</artifactId>
    <version>1.4</version>
</dependency>

<dependency>
    <groupId>org.apache.axis</groupId>
    <artifactId>axis-jaxrpc</artifactId>
    <version>1.4</version>
</dependency>

<dependency>
    <groupId>org.apache.axis</groupId>
    <artifactId>axis-saaj</artifactId>
    <version>1.4</version>
</dependency>

<dependency>
    <groupId>axis</groupId>
    <artifactId>axis-wsdl4j</artifactId>
    <version>1.5.1</version>
</dependency>

<dependency>
    <groupId>commons-discovery</groupId>
    <artifactId>commons-discovery</artifactId>
    <version>0.4</version>
</dependency>

<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
</dependency>
```

Then we need to configure maven to use Java 1.5 and the Axis maven plugin that generates the source code based on the wsdl file:

```
<!-- to compile with 1.5 -->
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.5</source>
        <target>1.5</target>
    </configuration>
</plugin>

<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>axis-tools-maven-plugin</artifactId>
    <configuration>
        <sourceDirectory>src/main/resources</sourceDirectory>
        <packageSpace>com.mycompany.myschema</packageSpace>
        <testCases>false</testCases>
        <serverSide>true</serverSide>
        <subPackageByFileName>false</subPackageByFileName>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>wsdl2java</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

```

        </executions>
    </plugin>
}

```

wsdl

We use the same .wsdl file as the Tutorial-Example-ReportIncident and copy it to `src/main/webapp/WEB-INF/wsdl`

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://reportincident.example.camel.apache.org"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://reportincident.example.camel.apache.org">

    <!-- Type definitions for input- and output parameters for webservice -->
    <wsdl:types>
        <xss:schema targetNamespace="http://reportincident.example.camel.apache.org">
            <xss:element name="inputReportIncident">
                <xss:complexType>
                    <xss:sequence>
                        <xss:element type="xs:string" name="incidentId"/>
                        <xss:element type="xs:string" name="incidentDate"/>
                        <xss:element type="xs:string" name="givenName"/>
                        <xss:element type="xs:string" name="familyName"/>
                        <xss:element type="xs:string" name="summary"/>
                        <xss:element type="xs:string" name="details"/>
                        <xss:element type="xs:string" name="email"/>
                        <xss:element type="xs:string" name="phone"/>
                    </xss:sequence>
                </xss:complexType>
            </xss:element>
            <xss:element name="outputReportIncident">
                <xss:complexType>
                    <xss:sequence>
                        <xss:element type="xs:string" name="code"/>
                    </xss:sequence>
                </xss:complexType>
            </xss:element>
        </xss:schema>
    </wsdl:types>

    <!-- Message definitions for input and output -->
    <wsdl:message name="inputReportIncident">
        <wsdl:part name="parameters" element="tns:inputReportIncident"/>
    </wsdl:message>
    <wsdl:message name="outputReportIncident">
        <wsdl:part name="parameters" element="tns:outputReportIncident"/>
    </wsdl:message>

    <!-- Port (interface) definitions -->
    <wsdl:portType name="ReportIncidentEndpoint">
        <wsdl:operation name="ReportIncident">
            <wsdl:input message="tns:inputReportIncident"/>
            <wsdl:output message="tns:outputReportIncident"/>
        </wsdl:operation>
    </wsdl:portType>

    <!-- Port bindings to transports and encoding - HTTP, document literal encoding is used -->
    <wsdl:binding name="ReportIncidentBinding" type="tns:ReportIncidentEndpoint">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="ReportIncident">
            <soap:operation
                soapAction="http://reportincident.example.camel.apache.org/ReportIncident"
                style="document"/>
            <wsdl:input>
                <soap:body parts="parameters" use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body parts="parameters" use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>

    <!-- Service definition -->
    <wsdl:service name="ReportIncidentService">
        <wsdl:port name="ReportIncidentPort" binding="tns:ReportIncidentBinding">
            <soap:address location="http://reportincident.example.camel.apache.org"/>
        </wsdl:port>
    </wsdl:service>

```

Configuring Axis

Okay we are now setup for the contract first development and can generate the source file. For now we are still only using standard Axis and not Spring nor Camel. We still need to setup Axis as a web application so we configure the `web.xml` in `src/main/webapp/WEB-INF/web.xml` as:

```

<servlet>
    <servlet-name>axis</servlet-name>
    <servlet-class>org.apache.axis.transport.http.AxisServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>axis</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>

```

The web.xml just registers Axis servlet that is handling the incoming web requests to its servlet mapping. We still need to configure Axis itself and this is done using its special configuration file server-config.wsdd. We nearly get this file for free if we let Axis generate the source code so we run the maven goal:

```
mvn axis-tools:wsdl2java
```

The tool will generate the source code based on the wsdl and save the files to the following folder:

```
.\target\generated-sources\axis-tools\wsdl2java\org\apache\camel\example\reportincident
deploy.wsdd
InputReportIncident.java
OutputReportIncident.java
ReportIncidentBindingImpl.java
ReportIncidentBindingStub.java
ReportIncidentService_PortType.java
ReportIncidentService_Service.java
ReportIncidentService_ServiceLocator.java
undeploy.wsdd
```

This is standard Axis and so far no Camel or Spring has been touched. To implement our webservice we will add our code, so we create a new class AxisReportIncidentService that implements the port type interface where we can implement our code logic what happens when the webservice is invoked.

```
package org.apache.camel.example.axis;

import org.apache.camel.example.reportincident.InputReportIncident;
import org.apache.camel.example.reportincident.OutputReportIncident;
import org.apache.camel.example.reportincident.ReportIncidentService_PortType;

import java.rmi.RemoteException;

/**
 * Axis webservice
 */
public class AxisReportIncidentService implements ReportIncidentService_PortType {

    public OutputReportIncident reportIncident(InputReportIncident parameters) throws RemoteException {
        System.out.println("Hello AxisReportIncidentService is called from " + parameters.getGivenName());
        OutputReportIncident out = new OutputReportIncident();
        out.setCode("OK");
        return out;
    }
}
```

Now we need to configure Axis itself and this is done using its server-config.wsdd file. We nearly get this for for free from the auto generated code, we copy the stuff from deploy.wsdd and made a few modifications:

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/" xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <!-- global configuration -->
  <globalConfiguration>
    <parameter name="sendXsiTypes" value="true"/>
    <parameter name="sendMultiRefs" value="true"/>
    <parameter name="sendXMLDeclaration" value="true"/>
    <parameter name="axis.sendMinimizedElements" value="true"/>
  </globalConfiguration>
  <handler name="URLMapper" type="java:org.apache.axis.handlers.http.URLMapper"/>

  <!-- this service is from deploy.wsdd -->
  <service name="ReportIncidentPort" provider="java:RPC" style="document" use="literal">
    <parameter name="wsdlTargetNamespace" value="http://reportincident.example.camel.apache.org"/>
    <parameter name="wsdlServiceElement" value="ReportIncidentService"/>
    <parameter name="wsdlServiceUnqualified" value="http://reportincident.example.camel.apache.org"/>
    <parameter name="wsdlServicePort" value="ReportIncidentPort"/>
    <parameter name="className" value="org.apache.camel.example.reportincident.ReportIncidentBindingImpl"/>
    <parameter name="wsdlPortType" value="ReportIncidentService"/>
    <parameter name="typeMappingVersion" value="1.2"/>
    <operation name="reportIncident" qname="ReportIncident" returnQName="retNS:outputReportIncident" xmlns:retNS="http://reportincident.example.camel.apache.org"
      returnType="rtns:outputReportIncident" xmlns:rtns="http://reportincident.example.camel.apache.org"
      soapAction="http://reportincident.example.camel.apache.org/ReportIncident" />
    <parameter qname="pns:inputReportIncident" xmlns:pns="http://reportincident.example.camel.apache.org"
      type="tns:inputReportIncident" xmlns:tns="http://reportincident.example.camel.apache.org"/>
  </operation>
  <parameter name="allowedMethods" value="reportIncident"/>

  <typeMapping
    xmlns:ns="http://reportincident.example.camel.apache.org"
    qname="ns:outputReportIncident"
    type="java:org.apache.camel.example.reportincident.OutputReportIncident"
    serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
    deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
    encodingStyle="" />
  <typeMapping
    xmlns:ns="http://reportincident.example.camel.apache.org"
    qname="ns:inputReportIncident"
    type="java:org.apache.camel.example.reportincident.InputReportIncident"
    serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
    deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
    encodingStyle="" />
</service>
<!-- part of Axis configuration -->
<transport name="http">
  <requestFlow>
```

```

<handler type="URLMapper"/>
<handler type="java:org.apache.axis.handlers.http.HTTPAuthHandler"/>
</requestFlow>
</transport>
</deployment>

```

The **globalConfiguration** and **transport** is not in the deploy.wsdd file so you gotta write that yourself. The **service** is a 100% copy from deploy.wsdd. Axis has more configuration to it than shown here, but then you should check the Axis documentation.

What we need to do now is important, as we need to modify the above configuration to use our webservice class than the default one, so we change the classname parameter to our class **AxisReportIncidentService**:

```
<parameter name="className" value="org.apache.camel.example.axis.AxisReportIncidentService"/>
```

Running the Example

Now we are ready to run our example for the first time, so we use Jetty as the quick web container using its maven command:

```
mvn jetty:run
```

Then we can hit the web browser and enter this URL: <http://localhost:8080/camel-example-axis/services> and you should see the famous Axis start page with the text **And now... Some Services**.

Clicking on the .wsdl link shows the wsdl file, but what. It's an auto generated one and not our original .wsdl file. So we need to fix this ASAP and this is done by configuring Axis in the server-config.wsdd file:

```

<service name="ReportIncidentPort" provider="java:RPC" style="document" use="literal">
  <wsdlFile>/WEB-INF/wsdl/report_incident.wsdl</wsdlFile>
  ...

```

We do this by adding the wsdlFile tag in the service element where we can point to the real .wsdl file.

Integrating Spring

First we need to add its dependencies to the **pom.xml**.

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>2.5.5</version>
</dependency>

```

Spring is integrated just as it would like to, we add its listener to the web.xml and a context parameter to be able to configure precisely what spring xml files to use:

```

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:axis-example-context.xml
  </param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

```

Next is to add a plain spring XML file named **axis-example-context.xml** in the src/main/resources folder.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
</beans>

```

The spring XML file is currently empty. We hit jetty again with `mvn jetty:run` just to make sure Spring was setup correctly.

Using Spring

We would like to be able to get hold of the Spring ApplicationContext from our webservice so we can get access to the glory spring, but how do we do this? And our webservice class AxisReportIncidentService is created and managed by Axis we want to let Spring do this. So we have two problems.

We solve these problems by creating a delegate class that Axis creates, and this delegate class gets hold on Spring and then gets our real webservice as a spring bean and invoke the service.

First we create a new class that is 100% independent from Axis and just a plain POJO. This is our real service.

```
package org.apache.camel.example.axis;
```

```

import org.apache.camel.example.reportincident.InputReportIncident;
import org.apache.camel.example.reportincident.OutputReportIncident;

/**
 * Our real service that is not tied to Axis
 */
public class ReportIncidentService {

    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        System.out.println("Hello ReportIncidentService is called from " + parameters.getGivenName());

        OutputReportIncident out = new OutputReportIncident();
        out.setCode("OK");
        return out;
    }
}

```

So now we need to get from AxisReportIncidentService to this one ReportIncidentService using Spring. Well first of all we add our real service to spring XML configuration file so Spring can handle its lifecycle:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="incidentservice" class="org.apache.camel.example.axis.ReportIncidentService"/>

</beans>

```

And then we need to modify AxisReportIncidentService to use Spring to lookup the spring bean **id="incidentservice"** and delegate the call. We do this by extending the spring class **org.springframework.remoting.jaxrpc.ServletEndpointSupport** so the refactored code is:

```

package org.apache.camel.example.axis;

import org.apache.camel.example.reportincident.InputReportIncident;
import org.apache.camel.example.reportincident.OutputReportIncident;
import org.apache.camel.example.reportincident.ReportIncidentService_PortType;
import org.springframework.remoting.jaxrpc.ServletEndpointSupport;

import java.rmi.RemoteException;

/**
 * Axis webservice
 */
public class AxisReportIncidentService extends ServletEndpointSupport implements ReportIncidentService_PortType {

    public OutputReportIncident reportIncident(InputReportIncident parameters) throws RemoteException {
        // get hold of the spring bean from the application context
        ReportIncidentService service = (ReportIncidentService) getApplicationContext().getBean("incidentservice");

        // delegate to the real service
        return service.reportIncident(parameters);
    }
}

```

To see if everything is okay we run `mvn jetty:run`.

In the code above we get hold of our service at each request by looking up in the application context. However Spring also supports an **init** method where we can do this once. So we change the code to:

```

public class AxisReportIncidentService extends ServletEndpointSupport implements ReportIncidentService_PortType {

    private ReportIncidentService service;

    @Override
    protected void onInit() throws ServiceException {
        // get hold of the spring bean from the application context
        service = (ReportIncidentService) getApplicationContext().getBean("incidentservice");
    }

    public OutputReportIncident reportIncident(InputReportIncident parameters) throws RemoteException {
        // delegate to the real service
        return service.reportIncident(parameters);
    }
}

```

So now we have integrated Axis with Spring and we are ready for Camel.

Integrating Camel

Again the first step is to add the dependencies to the maven **pom.xml** file:

```

<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-core</artifactId>
    <version>1.5.0</version>
</dependency>

<dependency>
    <groupId>org.apache.camel</groupId>

```

```
<artifactId>camel-spring</artifactId>
<version>1.5.0</version>
</dependency>
```

Now that we have integrated with Spring then we easily integrate with Camel as Camel works well with Spring.

Camel does not require Spring

Camel does not require Spring, we could easily have used Camel without Spring, but most users prefer to use Spring also.

We choose to integrate Camel in the Spring XML file so we add the camel namespace and the schema location:

```
xmlns:camel="http://activemq.apache.org/camel/schema/spring"
http://activemq.apache.org/camel/schema/spring http://activemq.apache.org/camel/schema/spring/camel-spring.xsd"
```

CamelContext

CamelContext is the heart of Camel its where all the routes, endpoints, components, etc. is registered. So we setup a CamelContext and the spring XML files looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://activemq.apache.org/camel/schema/spring"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://activemq.apache.org/camel/schema/spring http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">

    <bean id="incidentService" class="org.apache.camel.example.axis.ReportIncidentService"/>

    <camel:camelContext id="camel">
        <!-- TODO: Here we can add Camel stuff -->
    </camel:camelContext>
</beans>
```

Store a file backup

We want to store the web service request as a file before we return a response. To do this we want to send the file content as a message to an endpoint that produces the file. So we need to do two steps:

- configure the file backup endpoint
- send the message to the endpoint

The endpoint is configured in spring XML so we just add it as:

```
<camel:camelContext id="camelContext">
    <!-- endpoint named backup that is configured as a file component -->
    <camel:endpoint id="backup" uri="file:///target?append=false"/>
</camel:camelContext>
```

In the CamelContext we have defined our endpoint with the id backup and configured it use the URL notation that we know from the internet. Its a file scheme that accepts a context and some options. The context is target and its the folder to store the file. The option is just as the internet with ? and & for subsequent options. We configure it to not append, meaning than any existing file will be overwritten. See the File component for options and how to use the camel file endpoint.

Next up is to be able to send a message to this endpoint. The easiest way is to use a ProducerTemplate. A ProducerTemplate is inspired by Spring template pattern with for instance JmsTemplate or JdbcTemplate in mind. The template that all the grunt work and exposes a simple interface to the end-user where he/she can set the payload to send. Then the template will do proper resource handling and all related issues in that regard. But how do we get hold of such a template? Well the CamelContext is able to provide one. This is done by configuring the template on the camel context in the spring XML as:

```
<camel:camelContext id="camelContext">
    <!-- producer template exposed with this id -->
    <camel:template id="camelTemplate"/>

    <!-- endpoint named backup that is configured as a file component -->
    <camel:endpoint id="backup" uri="file:///target?append=false"/>
</camel:camelContext>
```

Then we can expose a ProducerTemplate property on our service with a setter in the Java code as:

```
public class ReportIncidentService {
    private ProducerTemplate template;
    public void setTemplate(ProducerTemplate template) {
        this.template = template;
    }
}
```

And then let Spring handle the dependency inject as below:

```
<bean id="incidentService" class="org.apache.camel.example.axis.ReportIncidentService">
    <!-- set the producer template to use from the camel context below -->
    <property name="template" ref="camelTemplate"/>
```

```
</bean>
```

Now we are ready to use the producer template in our service to send the payload to the endpoint. The template has many `sendXXX` methods for this purpose. But before we send the payload to the file endpoint we must also specify what filename to store the file as. This is done by sending meta data with the payload. In Camel metadata is sent as headers. Headers is just a plain `Map<String, Object>`. So if we needed to send several metadata then we could construct an ordinary `HashMap` and put the values in there. But as we just need to send one header with the filename Camel has a convenient send method `sendBodyAndHeader` so we choose this one.

```
public OutputReportIncident reportIncident(InputReportIncident parameters) {
    System.out.println("Hello ReportIncidentService is called from " + parameters.getGivenName());
    String data = parameters.getDetails();
    // store the data as a file
    String filename = parameters.getIncidentId() + ".txt";
    // send the data to the endpoint and the header contains what filename it should be stored as
    template.sendBodyAndHeader("backup", data, "org.apache.camel.file.name", filename);
    OutputReportIncident out = new OutputReportIncident();
    out.setCode("OK");
    return out;
}
```

The template in the code above uses 4 parameters:

- the endpoint name, in this case the id referring to the endpoint defined in Spring XML in the `camelContext` element.
- the payload, can be any kind of object
- the key for the header, in this case a Camel keyword to set the filename
- and the value for the header

Running the example

We start our integration with maven using `mvn jetty:run`. Then we open a browser and hit `http://localhost:8080`. Jetty is so smart that it display a frontpage with links to the deployed application so just hit the link and you get our application. Now we hit `/services` to the URL to access the Axis frontpage. The URL should be `http://localhost:8080/camel-example-axis/services`.

You can then test it using a web service test tools such as SoapUI.
Hitting the service will output to the console

```
2008-09-06 15:01:41.718::INFO: Started SelectChannelConnector @ 0.0.0.0:8080
[INFO] Started Jetty Server
Hello ReportIncidentService is called from Ibsen
```

And there should be a file in the target subfolder.

```
dir target /b
123.txt
```

Unit Testing

We would like to be able to unit test our `ReportIncidentService` class. So we add junit to the maven dependency:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.2</version>
    <scope>test</scope>
</dependency>
```

And then we create a plain junit testcase for our service class.

```
package org.apache.camel.example.axis;

import junit.framework.TestCase;
import org.apache.camel.example.reportincident.InputReportIncident;
import org.apache.camel.example.reportincident.OutputReportIncident;

/**
 * Unit test of service
 */
public class ReportIncidentServiceTest extends TestCase {

    public void testIncident() {
        ReportIncidentService service = new ReportIncidentService();

        InputReportIncident input = createDummyIncident();
        OutputReportIncident output = service.reportIncident(input);
        assertEquals("OK", output.getCode());
    }

    protected InputReportIncident createDummyIncident() {
        InputReportIncident input = new InputReportIncident();
        input.setEmail("davsclaus@apache.org");
    }
}
```

```

        input.setIncidentId("12345678");
        input.setIncidentDate("2008-07-13");
        input.setPhone("+45 2962 7576");
        input.setSummary("Failed operation");
        input.setDetails("The wrong foot was operated.");
        input.setFamilyName("Ibsen");
        input.setGivenName("Claus");
    }
}

```

Then we can run the test with maven using: mvn test. But we will get a failure:

```

Running org.apache.camel.example.axis.ReportIncidentServiceTest
Hello ReportIncidentService is called from Claus
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: 0.235 sec <<< FAILURE!
Results :
Tests in error:
testIncident(org.apache.camel.example.axis.ReportIncidentServiceTest)
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0

```

What is the problem? Well our service uses a CamelProducer (the template) to send a message to the file endpoint so the message will be stored in a file. What we need is to get hold of such a producer and inject it on our service, by calling the setter.

Since Camel is very light weight and embedable we are able to create a CamelContext and add the endpoint in our unit test code directly. We do this to show how this is possible:

```

private CamelContext context;

@Override
protected void setUp() throws Exception {
    super.setUp();
    // CamelContext is just created like this
    context = new DefaultCamelContext();

    // then we can create our endpoint and set the options
    FileEndpoint endpoint = new FileEndpoint();
    // the endpoint must have the camel context set also
    endpoint.setCamelContext(context);
    // our output folder
    endpoint.setFile(new File("target"));
    // and the option not to append
    endpoint.setAppend(false);

    // then we add the endpoint just in java code just as the spring XML, we register it with the "backup" id.
    context.addSingletonEndpoint("backup", endpoint);

    // finally we need to start the context so Camel is ready to rock
    context.start();
}

@Override
protected void tearDown() throws Exception {
    super.tearDown();
    // and we are nice boys so we stop it to allow resources to clean up
    context.stop();
}

```

So now we are ready to set the ProducerTemplate on our service, and we get a hold of that baby from the CamelContext as:

```

public void testIncident() {
    ReportIncidentService service = new ReportIncidentService();

    // get a producer template from the camel context
    ProducerTemplate template = context.createProducerTemplate();
    // inject it on our service using the setter
    service.setTemplate(template);

    InputReportIncident input = createDummyIncident();
    OutputReportIncident output = service.reportIncident(input);
    assertEquals("OK", output.getCode());
}

```

And this time when we run the unit test its a success:

```

Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

```

We would like to test that the file exists so we add these two lines to our test method:

```

// should generate a file also
File file = new File("target/" + input.getIncidentId() + ".txt");
assertTrue("File should exists", file.exists());

```

Smarter Unit Testing with Spring

The unit test above requires us to assemble the Camel pieces manually in java code. What if we would like our unit test to use our spring configuration file **axis-example-context.xml** where we already have setup the endpoint. And of course we would like

to test using this configuration file as this is the real file we will use. Well hey presto the xml file is a spring ApplicationContext file and spring is able to load it, so we go the spring path for unit testing. First we add the spring-test jar to our maven dependency:

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-test</artifactId>
<scope>test</scope>
</dependency>
```

And then we refactor our unit test to be a standard spring unit class. What we need to do is to extend `AbstractJUnit3SpringContextTests` instead of `TestCase` in our unit test. Since Spring 2.5 embraces annotations we will use one as well to instruct what our xml configuration file is located:

```
@ContextConfiguration(locations = "classpath:axis-example-context.xml")
public class ReportIncidentServiceTest extends AbstractJUnit3SpringContextTests {
```

What we must remember to add is the `classpath:` prefix as our xml file is located in `src/main/resources`. If we omit the prefix then Spring will by default try to locate the xml file in the current package and that is `org.apache.camel.example.axis`. If the xml file is located outside the classpath you can use file: prefix instead. So with these two modifications we can get rid of all the setup and teardown code we had before and now we will test our real configuration.

The last change is to get hold of the producer template and now we can just refer to the bean id it has in the spring xml file:

```
<!-- producer template exposed with this id -->
<camel:template id="camelTemplate"/>
```

So we get hold of it by just getting it from the spring ApplicationContext as all spring users is used to do:

```
// get a producer template from the the spring context
ProducerTemplate template = (ProducerTemplate) applicationContext.getBean("camelTemplate");
// inject it on our service using the setter
service.setTemplate(template);
```

Now our unit test is much better, and a real power of Camel is that it fits nicely with Spring and you can use standard Spring'ish unit test to test your Camel applications as well.

Unit Test calling WebService

What if you would like to execute a unit test where you send a webservice request to the `AxisReportIncidentService` how do we unit test this one? Well first of all the code is merely just a delegate to our real service that we have just tested, but nevertheless its a good question and we would like to know how. Well the answer is that we can exploit that fact that Jetty is also a slim web container that can be embedded anywhere just as Camel can. So we add this to our pom.xml:

```
<dependency>
<groupId>org.mortbay.jetty</groupId>
<artifactId>jetty</artifactId>
<version>${jetty-version}</version>
<scope>test</scope>
</dependency>
```

Then we can create a new class `AxisReportIncidentServiceTest` to unit test with Jetty. The code to setup Jetty is shown below with code comments:

```
public class AxisReportIncidentServiceTest extends TestCase {
    private Server server;
    private void startJetty() throws Exception {
        // create an embedded Jetty server
        server = new Server();
        // add a listener on port 8080 on localhost (127.0.0.1)
        Connector connector = new SelectChannelConnector();
        connector.setPort(8080);
        connector.setHost("127.0.0.1");
        server.addConnector(connector);
        // add our web context path
        WebAppContext wac = new WebAppContext();
        wac.setContextPath("/unittest");
        // set the location of the exploded webapp where WEB-INF is located
        // this is a nice feature of Jetty where we can point to src/main/webapp
        wac.setWar("./src/main/webapp");
        server.setHandler(wac);
        // then start Jetty
        server.setStopAtShutdown(true);
        server.start();
    }
    @Override
    protected void setUp() throws Exception {
        super.setUp();
        startJetty();
    }
    @Override
    protected void tearDown() throws Exception {
```

```

        super.tearDown();
        server.stop();
    }
}

```

Now we just need to send the incident as a webservice request using Axis. So we add the following code:

```

public void testReportIncidentWithAxis() throws Exception {
    // the url to the axis webservice exposed by jetty
    URL url = new URL("http://localhost:8080/unittest/services/ReportIncidentPort");

    // Axis stub to get the port where we can send the webservice request
    ReportIncidentService_ServiceLocator locator = new ReportIncidentService_ServiceLocator();
    ReportIncidentService_PortType port = locator.getReportIncidentPort(url);

    // create input to send
    InputReportIncident input = createDummyIncident();
    // send the webservice and get the response
    OutputReportIncident output = port.reportIncident(input);
    assertEquals("OK", output.getCode());

    // should generate a file also
    File file = new File("target/" + input.getIncidentId() + ".txt");
    assertTrue("File should exists", file.exists());
}

protected InputReportIncident createDummyIncident() {
    InputReportIncident input = new InputReportIncident();
    input.setEmailAddress("davclaus@apache.org");
    input.setIncidentId("12345678");
    input.setIncidentDate("2008-07-13");
    input.setPhone("+45 2962 7576");
    input.setSummary("Failed operation");
    input.setDetails("The wrong foot was operated.");
    input.setFamilyName("Ibsen");
    input.setGivenName("Claus");
    return input;
}

```

And now we have an unittest that sends a webservice request using good old Axis.

Annotations

Both Camel and Spring has annotations that can be used to configure and wire trivial settings more elegantly. Camel has the endpoint annotation `@EndpointInjected` that is just what we need. With this annotation we can inject the endpoint into our service. The annotation takes either a name or uri parameter. The name is the bean id in the Registry. The uri is the URI configuration for the endpoint. Using this you can actually inject an endpoint that you have not defined in the camel context. As we have defined our endpoint with the id `backup` we use the name parameter.

```

@EndpointInject(name = "backup")
private ProducerTemplate template;

```

Camel is smart as `@EndpointInjected` supports different kinds of object types. We like the `ProducerTemplate` so we just keep it as it is.

Since we use annotations on the field directly we do not need to set the property in the spring xml file so we change our service bean:

```

<bean id="incidentservice" class="org.apache.camel.example.axis.ReportIncidentService"/>

```

Running the unit test with `mvn test` reveals that it works nicely.

And since we use the `@EndpointInjected` that refers to the endpoint with the id `backup` directly we can loose the template tag in the xml, so its shorter:

```

<bean id="incidentservice" class="org.apache.camel.example.axis.ReportIncidentService"/>

<camel:camelContext id="camelContext">
    <!-- producer template exposed with this id -->
    <camel:template id="camelTemplate">
        <!-- endpoint named backup that is configured as a file component -->
        <camel:endpoint id="backup" uri="file:///target/append=false"/>
    </camel:template>
</camel:camelContext>

```

And the final touch we can do is that since the endpoint is injected with concrete endpoint to use we can remove the "backup" name parameter when we send the message. So we change from:

```

// send the data to the endpoint and the header contains what filename it should be stored as
template.sendBodyAndHeader("backup", data, "org.apache.camel.file.name", filename);

```

To without the name:

```

// send the data to the endpoint and the header contains what filename it should be stored as
template.sendBodyAndHeader(data, "org.apache.camel.file.name", filename);

```

Then we avoid to duplicate the name and if we rename the endpoint name then we don't forget to change it in the code also.

The End

This tutorial hasn't really touched the one of the key concept of Camel as a powerful routing and mediation framework. But we wanted to demonstrate its flexibility and that it integrates well with even older frameworks such as Apache Axis 1.4.

Check out the other tutorials on Camel and the other examples.

Note that the code shown here also applies to Camel 1.4 so actually you can get started right away with the released version of Camel. As this time of writing Camel 1.5 is work in progress.

See Also

- Tutorials
- Examples

TUTORIAL ON USING CAMEL IN A WEB APPLICATION

Camel has been designed to work great with the Spring framework; so if you are already a Spring user you can think of Camel as just a framework for adding to your Spring XML files.

So you can follow the usual Spring approach to working with web applications; namely to add the standard Spring hook to load a **/WEB-INF/applicationContext.xml** file. In that file you can include your usual Camel XML configuration.

Step1: Edit your web.xml

To enable spring add a context loader listener to your **/WEB-INF/web.xml** file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
          version="2.5">

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

</web-app>
```

This will cause Spring to boot up and look for the **/WEB-INF/applicationContext.xml** file.

Step 2: Create a /WEB-INF/applicationContext.xml file

Now you just need to create your Spring XML file and add your camel routes or configuration.

For example

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd
                           http://camel.apache.org/schema/spring
                           http://camel.apache.org/schema/spring/camel-spring.xsd">

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="seda:foo"/>
            <to uri="mock:results"/>
        </route>
    </camelContext>
</beans>
```

Then boot up your web application and you're good to go!

Hints and Tips

If you use Maven to build your application your directory tree will look like this...

```
src/main/webapp/WEB-INF
  web.xml
  applicationContext.xml
```

You should update your Maven pom.xml to enable WAR packaging/naming like this...

```
<project>
  ...
  <packaging>war</packaging>
  <build>
    <finalName>[desired WAR file name]</finalName>
    ...
  </build>
```

To enable more rapid development we highly recommend the jetty:run maven plugin.

Please refer to the help for more information on using jetty:run - but briefly if you add the following to your pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty-plugin</artifactId>
      <configuration>
        <webAppConfig>
          <contextPath>/</contextPath>
        </webAppConfig>
        <scanIntervalSeconds>10</scanIntervalSeconds>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Then you can run your web application as follows

```
mvn jetty:run
```

Then Jetty will also monitor your target/classes directory and your src/main/webapp directory so that if you modify your spring XML, your web.xml or your java code the web application will be restarted, re-creating your Camel routes.

If your unit tests take a while to run, you could miss them out when running your web application via

```
mvn -Dtest=false jetty:run
```

TUTORIAL BUSINESS PARTNERS

Under Construction

This tutorial is a work in progress.

BACKGROUND AND INTRODUCTION

Business Background

So there's a company, which we'll call Acme. Acme sells widgets, in a fairly unusual way. Their customers are responsible for telling Acme what they purchased. The customer enters into their own systems (ERP or whatever) which widgets they bought from Acme. Then at some point, their systems emit a record of the sale which needs to go to Acme so Acme can bill them for it. Obviously, everyone wants this to be as automated as possible, so there needs to be integration between the customer's system and Acme.

Sadly, Acme's sales people are, technically speaking, doormats. They tell all their prospects, "you can send us the data in whatever format, using whatever protocols, whatever. You just can't change once it's up and running."

The result is pretty much what you'd expect. Taking a random sample of 3 customers:

- Customer 1: **XML over FTP**
- Customer 2: **CSV over HTTP**
- Customer 3: **Excel via e-mail**

Now on the Acme side, all this has to be converted to a canonical XML format and submitted to the Acme accounting system via JMS. Then the Acme accounting system does its stuff and sends an XML reply via JMS, with a summary of what it processed (e.g. 3 line items accepted, line item #2 in error, total invoice \$123.45). Finally, that data needs to be formatted into an e-mail, and sent to a contact at the customer in question ("Dear Joyce, we received an invoice on 1/2/08. We accepted 3 line items totaling \$123.45, though there was an error with line items #2 [invalid quantity ordered]. Thank you for your business. Love, Acme.").

So it turns out Camel can handle all this:

- Listen for HTTP, e-mail, and FTP files
- Grab attachments from the e-mail messages
- Convert XML, XLS, and CSV files to a canonical XML format
- read and write JMS messages
- route based on company ID
- format e-mails using Velocity templates
- send outgoing e-mail messages

Tutorial Background

This tutorial will cover all that, plus setting up tests along the way.

Before starting, you should be familiar with:

- Camel concepts including the CamelContext, Routes, Components and Endpoints, and Enterprise Integration Patterns
- Configuring Camel with the XML or Java DSL

You'll learn:

- How to set up a Maven build for a Camel project
- How to transform XML, CSV, and Excel data into a standard XML format with Camel
 - How to write POJOs (Plain Old Java Objects), Velocity templates, and XSLT stylesheets that are invoked by Camel routes for message transformation
- How to configure simple and complex Routes in Camel, using either the XML or the Java DSL format
- How to set up unit tests that load a Camel configuration and test Camel routes
- How to use Camel's Data Formats to automatically convert data between Java objects and XML, CSV files, etc.
- How to send and receive e-mail from Camel
- How to send and receive JMS messages from Camel
- How to use Enterprise Integration Patterns including Message Router and Pipes and Filters
 - How to use various languages to express content-based routing rules in Camel
- How to deal with Camel messages, headers, and attachments

You may choose to treat this as a hands-on tutorial, and work through building the code and configuration files yourself. Each of the sections gives detailed descriptions of the steps that need to be taken to get the components and routes working in Camel, and takes you through tests to make sure they are working as expected.

But each section also links to working copies of the source and configuration files, so if you don't want the hands-on approach, you can simply review and/or download the finished files.

High-Level Diagram

Here's more or less what the integration process looks like.

First, the input from the customers to Acme:



And then, the output from Acme to the customers:



Tutorial Tasks

To get through this scenario, we're going to break it down into smaller pieces, implement and test those, and then try to assemble the big scenario and test that.

Here's what we'll try to accomplish:

1. Create a Maven build for the project
2. Get sample files for the customer Excel, CSV, and XML input
3. Get a sample file for the canonical XML format that Acme's accounting system uses
4. Create an XSD for the canonical XML format
5. Create JAXB POJOs corresponding to the canonical XSD
6. Create an XSLT stylesheet to convert the Customer 1 (XML over FTP) messages to the canonical format
7. Create a unit test to ensure that a simple Camel route invoking the XSLT stylesheet works
8. Create a POJO that converts a `List<List<String>>` to the above JAXB POJOs
 - Note that Camel can automatically convert CSV input to a List of Lists of Strings representing the rows and columns of the CSV, so we'll use this POJO to handle Customer 2 (CSV over HTTP)
9. Create a unit test to ensure that a simple Camel route invoking the CSV processing works
10. Create a POJO that converts a Customer 3 Excel file to the above JAXB POJOs (using POI to read Excel)
11. Create a unit test to ensure that a simple Camel route invoking the Excel processing works
12. Create a POJO that reads an input message, takes an attachment off the message, and replaces the body of the message with the attachment
 - This is assuming for Customer 3 (Excel over e-mail) that the e-mail contains a single Excel file as an attachment, and the actual e-mail body is throwaway
13. Build a set of Camel routes to handle the entire input (Customer -> Acme) side of the scenario.
14. Build unit tests for the Camel input.
15. **TODO:** Tasks for the output (Acme -> Customer) side of the scenario

LET'S GET STARTED!

Step 1: Initial Maven build

We'll use Maven for this project as there will eventually be quite a few dependencies and it's nice to have Maven handle them for us. You should have a current version of Maven (e.g. 2.0.9) installed.

You can start with a pretty empty project directory and a Maven POM file, or use a simple JAR archetype to create one.

Here's a sample POM. We've added a dependency on **camel-core**, and set the compile version to 1.5 (so we can use annotations):

```
Listing 1. pom.xml
<?xml version="1.0" encoding="UTF-8"?>
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.apache.camel.tutorial</groupId>
    <artifactId>business-partners</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Camel Business Partners Tutorial</name>
    <dependencies>
        <dependency>
            <artifactId>camel-core</artifactId>
            <groupId>org.apache.camel</groupId>
            <version>1.4.0</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Step 2: Get Sample Files

You can make up your own if you like, but here are the "off the shelf" ones. You can save yourself some time by downloading these to `src/test/resources` in your Maven project.

- Customer 1 (XML): `input-customer1.xml`
- Customer 2 (CSV): `input-customer2.csv`
- Customer 3 (Excel): `input-customer3.xls`
- Canonical Acme XML Request: `canonical-acme-request.xml`
- Canonical Acme XML Response: `TODO`

If you look at these files, you'll see that the different input formats use different field names and/or ordering, because of course the sales guys were totally OK with that. Sigh.

Step 3: XSD and JAXB Beans for the Canonical XML Format

Here's the sample of the canonical XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice xmlns="http://activemq.apache.org/camel/tutorial/partners/invoice">
    <partner-id>2</partner-id>
    <date-received>6/12/2008</date-received>
    <line-item>
        <product-id>134</product-id>
        <description>widget</description>
        <quantity>3</quantity>
        <item-price>10.45</item-price>
        <order-date>6/5/2008</order-date>
    </line-item>
    <!-- // more line-item elements here -->
    <order-total>218.82</order-total>
</invoice>
```

If you're ambitious, you can write your own XSD (XML Schema) for files that look like this, and save it to `src/main/xsd`.

Solution: If not, you can download mine, and save that to save it to `src/main/xsd`.

Generating JAXB Beans

Down the road we'll want to deal with the XML as Java POJOs. We'll take a moment now to set up those XML binding POJOs. So we'll update the Maven POM to generate JAXB beans from the XSD file.

We need a dependency:

```
<dependency>
    <artifactId>camel-jaxb</artifactId>
    <groupId>org.apache.camel</groupId>
    <version>1.4.0</version>
</dependency>
```

And a plugin configured:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxb2-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>xjc</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

That should do it (it automatically looks for XML Schemas in `src/main/xsd` to generate beans for). Run `mvn install` and it should emit the beans into `target/generated-sources/jaxb`. Your IDE should see them there, though you may need to update the project to reflect the new settings in the Maven POM.

Step 4: Initial Work on Customer 1 Input (XML over FTP)

To get a start on Customer 1, we'll create an XSLT template to convert the Customer 1 sample file into the canonical XML format, write a small Camel route to test it, and build that into a unit test. If we get through this, we can be pretty sure that the XSLT template is valid and can be run safely in Camel.

Create an XSLT template

Start with the Customer 1 sample input. You want to create an XSLT template to generate XML like the canonical XML sample above – an `invoice` element with `line-item` elements (one per item in the original XML document). If you're especially clever, you can populate the current date and order total elements too.

Solution: My sample XSLT template isn't that smart, but it'll get you going if you don't want to write one of your own.

Create a unit test

Here's where we get to some meaty Camel work. We need to:

- Set up a unit test
- That loads a Camel configuration
- That has a route invoking our XSLT
- Where the test sends a message to the route
- And ensures that some XML comes out the end of the route

The easiest way to do this is to set up a Spring context that defines the Camel stuff, and then use a base unit test class from Spring that knows how to load a Spring context to run tests against. So, the procedure is:

Set Up a Skeletal Camel/Spring Unit Test

1. Add dependencies on Camel-Spring, and the Spring test JAR (which will automatically bring in JUnit 3.8.x) to your POM:

```
<dependency>
  <artifactId>camel-spring</artifactId>
  <groupId>org.apache.camel</groupId>
  <version>1.4.0</version>
</dependency>
<dependency>
  <artifactId>spring-test</artifactId>
  <groupId>org.springframework</groupId>
  <version>2.5.3</version>
  <scope>test</scope>
</dependency>
```

2. Create a new unit test class in `src/test/java/your-package-here`, perhaps called `XMLInputTest.java`
3. Make the test extend Spring's `AbstractJUnit38SpringContextTests` class, so it can load a Spring context for the test
4. Create a Spring context configuration file in `src/test/resources`, perhaps called `XMLInputTest-context.xml`
5. In the unit test class, use the class-level `@ContextConfiguration` annotation to indicate that a Spring context should be loaded
 - By default, this looks for a Context configuration file called `TestClassName-context.xml` in a subdirectory corresponding to the package of the test class. For instance, if your test class was `org.apache.camel.tutorial.XMLInputTest`, it would look for `org/apache/camel/tutorial/XMLInputTest-context.xml`
 - To override this default, use the `locations` attribute on the `@ContextConfiguration` annotation to provide specific context file locations (starting each path with a / if you don't want it to be relative to the package directory). My solution does this so I can put the context file directly in `src/test/resources` instead of in a package directory under there.
6. Add a `CamelContext` instance variable to the test class, with the `@Autowired` annotation. That way Spring will automatically pull the `CamelContext` out of the Spring context and inject it into our test class.
7. Add a `ProducerTemplate` instance variable and a `setup` method that instantiates it from the `CamelContext`. We'll use the `ProducerTemplate` later to send messages to the route.

```

protected ProducerTemplate<Exchange> template;
protected void setUp() throws Exception {
    super.setUp();
    template = camelContext.createProducerTemplate();
}

```

8. Put in an empty test method just for the moment (so when we run this we can see that "1 test succeeded")
 9. Add the Spring <beans> element (including the Camel Namespace) with an empty <camelContext> element to the Spring context, like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://activemq.apache.org/camel/schema/spring
                           http://activemq.apache.org/camel/schema/spring/camel-spring-1.4.0.xsd">
    <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
        </camelContext>
</beans>

```

Test it by running **mvn install** and make sure there are no build errors. So far it doesn't test much; just that your project and test and source files are all organized correctly, and the one empty test method completes successfully.

Solution: Your test class might look something like this:

- src/test/java/org/apache/camel/tutorial/XMLInputTest.java
- src/test/resources/XMLInputTest-context.xml (same as just above)

Flesh Out the Unit Test

So now we're going to write a Camel route that applies the XSLT to the sample Customer 1 input file, and makes sure that some XML output comes out:

- Save the input-customer1.xml file to src/test/resources
- Save your XSLT file (created in the previous step) to src/main/resources
- Write a Camel Route, either right in the Spring XML, or using the Java DSL (in another class under src/test/java somewhere). This route should use the Pipes and Filters integration pattern to:
 - Start from the endpoint direct:start (which lets the test conveniently pass messages into the route)
 - Call the endpoint xslt:YourXSLTFile.xsl (to transform the message with the specified XSLT template)
 - Send the result to the endpoint mock:finish (which lets the test verify the route output)
- Add a test method to the unit test class that:
 - Get a reference to the Mock endpoint mock:finish using code like this:

```
MockEndpoint finish = MockEndpoint.resolve(camelContext, "mock:finish");
```

- Set the expectedMessageCount on that endpoint to 1
- Get a reference to the Customer 1 input file, using code like this:

```
InputStream in = XMLInputTest.class.getResourceAsStream("/input-partner1.xml");
assertNotNull(in);
```

- Send that InputStream as a message to the direct:start endpoint, using code like this:

```
template.sendBody("direct:start", in);
```

Note that we can send the sample file body in several formats (File, InputStream, String, etc.) but in this case an InputStream is pretty convenient.

- Ensure that the message made it through the route to the final endpoint, by testing all configured Mock endpoints like this:

```
MockEndpoint.assertIsSatisfied(camelContext);
```

- If you like, inspect the final message body using some code like
`finish.getExchanges().get(0).getIn().getBody()`.
 - If you do this, you'll need to know what format that body is – String, byte array, InputStream, etc.
- Run your test with **mvn install** and make sure the build completes successfully.

Solution: Your finished test might look something like this:

- src/test/java/org/apache/camel/tutorial/XMLInputTest.java
- For XML Configuration:
 - src/test/resources/XMLInputTest-context.xml
- Or, for Java DSL Configuration:

- src/test/resources/XMLInputTest-dsl-context.xml
- src/test/java/org/apache/camel/tutorial/routes/XMLInputTestRoute.java

Test Base Class

Once your test class is working, you might want to extract things like the @Autowired CamelContext, the ProducerTemplate, and the setUp method to a custom base class that you extend with your other tests.

Step 5: Initial Work on Customer 2 Input (CSV over HTTP)

To get a start on Customer 2, we'll create a POJO to convert the Customer 2 sample CSV data into the JAXB POJOs representing the canonical XML format, write a small Camel route to test it, and build that into a unit test. If we get through this, we can be pretty sure that the CSV conversion and JAXB handling is valid and can be run safely in Camel.

Create a CSV-handling POJO

To begin with, CSV is a known data format in Camel. Camel can convert a CSV file to a List (representing rows in the CSV) of Lists (representing cells in the row) of Strings (the data for each cell). That means our POJO can just assume the data coming in is of type `List<List<String>>`, and we can declare a method with that as the argument.

Looking at the JAXB code in `target/generated-sources/jaxb`, it looks like an `Invoice` object represents the whole document, with a nested list of `LineItem` objects for the line items. Therefore our POJO method will return an `Invoice` (a document in the canonical XML format).

So to implement the CSV-to-JAXB POJO, we need to do something like this:

1. Create a new class under `src/main/java`, perhaps called `CSVConverterBean`.
2. Add a method, with one argument of type `List<List<String>>` and the return type `Invoice`
 - You may annotate the argument with `@Body` to specifically designate it as the body of the incoming message
3. In the method, the logic should look roughly like this:
 1. Create a new `Invoice`, using the method on the generated `ObjectFactory` class
 2. Loop through all the rows in the incoming CSV (the outer `List`)
 3. Skip the first row, which contains headers (column names)
 4. For the other rows:
 1. Create a new `LineItemType` (using the `ObjectFactory` again)
 2. Pick out all the cell values (the Strings in the inner `List`) and put them into the correct fields of the `LineItemType`
 - Not all of the values will actually go into the line item in this example
 - You may hardcode the column ordering based on the sample data file, or else try to read it dynamically from the headers in the first line
 - Note that you'll need to use a JAXB `datatypeFactory` to create the `XMLGregorianCalendar` values that JAXB uses for the date fields in the XML – which probably means using a `simpleDateFormat` to parse the date and setting that date on a `GregorianCalendar`
 3. Add the line item to the invoice
 5. Populate the partner ID, date of receipt, and order total on the `Invoice`
 6. Throw any exceptions out of the method, so Camel knows something went wrong
 7. Return the finished `Invoice`

Solution: Here's an example of what the `CSVConverterBean` might look like.

Create a unit test

Start with a simple test class and test Spring context like last time, perhaps based on the name `CSVInputTest`:

```
Listing 1. CSVInputTest.java

/**
 * A test class to ensure we can convert Partner 2 CSV input files to the
 * canonical XML output format, using JAXB POJOs.
 */
@ContextConfiguration(locations = "/CSVInputTest-context.xml")
public class CSVInputTest extends AbstractJUnit3SpringContextTests {
    @Autowired
    protected CamelContext camelContext;
    protected ProducerTemplate<Exchange> template;

    protected void setUp() throws Exception {
        super.setUp();
        template = camelContext.createProducerTemplate();
    }

    public void testCSVConversion() {
        // TODO
    }
}
```

```
Listing 1. CSVInputTest-context.xml

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xsi:schemaLocation="http://www.springframework.org/schema/beans
                    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                    http://activemq.apache.org/camel/schema/spring
                    http://activemq.apache.org/camel/schema/spring/camel-spring-1.4.0.xsd"
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
    <!-- TODO -->
</camelContext>
</beans>

```

Now the meaty part is to flesh out the test class and write the Camel routes.

1. Update the Maven POM to include CSV Data Format support:

```

<dependency>
    <artifactId>camel-csv</artifactId>
    <groupId>org.apache.camel</groupId>
    <version>1.4.0</version>
</dependency>

```

2. Write the routes (right in the Spring XML context, or using the Java DSL) for the CSV conversion process, again using the Pipes and Filters pattern:

1. Start from the endpoint direct:CSVstart (which lets the test conveniently pass messages into the route). We'll name this differently than the starting point for the previous test, in case you use the Java DSL and put all your routes in the same package (which would mean that each test would load the DSL routes for several tests.)
2. This time, there's a little preparation to be done. Camel doesn't know that the initial input is a CSV, so it won't be able to convert it to the expected `List<List<String>>` without a little hint. For that, we need an unmarshal transformation in the route. The `unmarshal` method (in the DSL) or `element` (in the XML) takes a child indicating the format to unmarshal; in this case that should be `csv`.
3. Next invoke the POJO to transform the message with a bean:CSVConverter endpoint
4. As before, send the result to the endpoint mock:finish (which lets the test verify the route output)
5. Finally, we need a Spring `<bean>` element in the Spring context XML file (but outside the `<camelContext>` element) to define the Spring bean that our route invokes. This Spring bean should have a `name` attribute that matches the name used in the bean endpoint (`csvConverter` in the example above), and a `class` attribute that points to the CSV-to-JAXB POJO class you wrote above (such as, `org.apache.camel.tutorial.CSVConverterBean`). When Spring is in the picture, any bean endpoints look up Spring beans with the specified name.
3. Write a test method in the test class, which should look very similar to the previous test class:
 1. Get the MockEndpoint for the final endpoint, and tell it to expect one message
 2. Load the Partner 2 sample CSV file from the ClassPath, and send it as the body of a message to the starting endpoint
 3. Verify that the final MockEndpoint is satisfied (that is, it received one message) and examine the message body if you like
 - Note that we didn't marshal the JAXB POJOs to XML in this test, so the final message should contain an `Invoice` as the body. You could write a simple line of code to get the Exchange (and Message) from the MockEndpoint to confirm that.
4. Run this new test with `mvn install` and make sure it passes and the build completes successfully.

Solution: Your finished test might look something like this:

- `src/test/java/org/apache/camel/tutorial/CSVInputTest.java`
- For XML Configuration:
 - `src/test/resources/CSVInputTest-context.xml`
- Or, for Java DSL Configuration:
 - `src/test/resources/CSVInputTest-dsl-context.xml`
 - `src/test/java/org/apache/camel/tutorial/routes/CSVInputTestRoute.java`

Step 6: Initial Work on Customer 3 Input (Excel over e-mail)

To get a start on Customer 3, we'll create a POJO to convert the Customer 3 sample Excel data into the JAXB POJOs representing the canonical XML format, write a small Camel route to test it, and build that into a unit test. If we get through this, we can be pretty sure that the Excel conversion and JAXB handling is valid and can be run safely in Camel.

Create an Excel-handling POJO

Camel does not have a data format handler for Excel by default. We have two options – create an Excel DataFormat (so Camel can convert Excel spreadsheets to something like the CSV `List<List<String>>` automatically), or create a POJO that can translate Excel data manually. For now, the second approach is easier (if we go the DataFormat route, we need code to both read and write Excel files, whereas otherwise read-only will do).

So, we need a POJO with a method that takes something like an `InputStream` or `byte[]` as an argument, and returns an `Invoice` as before. The process should look something like this:

1. Update the Maven POM to include POI support:

```

<dependency>
    <!-- POI -->

```

```

<artifactId>poi</artifactId>
<groupId>org.apache.poi</groupId>
<version>3.1-FINAL</version>
</dependency>

```

2. Create a new class under `src/main/java`, perhaps called `ExcelConverterBean`.
3. Add a method, with one argument of type `InputStream` and the return type `Invoice`
 - o You may annotate the argument with `@Body` to specifically designate it as the body of the incoming message
4. In the method, the logic should look roughly like this:
 1. Create a new `Invoice`, using the method on the generated `ObjectFactory` class
 2. Create a new `HSSFWorkbook` from the `InputStream`, and get the first sheet from it
 3. Loop through all the rows in the sheet
 4. Skip the first row, which contains headers (column names)
 5. For the other rows:
 1. Create a new `LineItemType` (using the `ObjectFactory` again)
 2. Pick out all the cell values and put them into the correct fields of the `LineItemType` (you'll need some data type conversion logic)
 - Not all of the values will actually go into the line item in this example
 - You may hardcode the column ordering based on the sample data file, or else try to read it dynamically from the headers in the first line
 - Note that you'll need to use a JAXB `datatypeFactory` to create the `XMLGregorianCalendar` values that JAXB uses for the date fields in the XML – which probably means setting the date from a date cell on a `GregorianCalendar`
 3. Add the line item to the invoice
 6. Populate the partner ID, date of receipt, and order total on the `Invoice`
 7. Throw any exceptions out of the method, so Camel knows something went wrong
 8. Return the finished `Invoice`

Solution: Here's an example of what the `ExcelConverterBean` might look like.

Create a unit test

The unit tests should be pretty familiar now. The test class and context for the Excel bean should be quite similar to the CSV bean.

1. Create the basic test class and corresponding Spring Context XML configuration file
2. The XML config should look a lot like the CSV test, except:
 - o Remember to use a different start endpoint name if you're using the Java DSL and not use separate packages per test
 - o You don't need the `unmarshal` step since the Excel POJO takes the raw `InputStream` from the source endpoint
 - o You'll declare a `<bean>` and endpoint for the Excel bean prepared above instead of the CSV bean
3. The test class should look a lot like the CSV test, except use the right input file name and start endpoint name.

Logging

You may notice that your tests emit a lot less output all of a sudden. The dependency on POI brought in Log4J and configured commons-logging to use it, so now we need a `log4j.properties` file to configure log output. You can use the attached one (snarfed from ActiveMQ) or write your own; either way save it to `src/main/resources` to ensure you continue to see log output.

Solution: Your finished test might look something like this:

- `src/test/java/org/apache/camel/tutorial/ExcelInputTest.java`
- For XML Configuration:
 - o `src/test/resources/ExcelInputTest-context.xml`
- Or, for Java DSL Configuration:
 - o `src/test/resources/ExcelInputTest-dsl-context.xml`
 - o `src/test/java/org/apache/camel/tutorial/routes/ExcelInputTestRoute.java`

Step 7: Put this all together into Camel routes for the Customer Input

With all the data type conversions working, the next step is to write the real routes that listen for HTTP, FTP, or e-mail input, and write the final XML output to an ActiveMQ queue. Along the way these routes will use the data conversions we've developed above.

So we'll create 3 routes to start with, as shown in the diagram back at the beginning:

1. Accept XML orders over FTP from Customer 1 (we'll assume the FTP server dumps files in a local directory on the Camel machine)
 2. Accept CSV orders over HTTP from Customer 2
 3. Accept Excel orders via e-mail from Customer 3 (we'll assume the messages are sent to an account we can access via IMAP)
- ...

Step 8: Create a unit test for the Customer Input Routes

Languages Supported Appendix

To support flexible and powerful Enterprise Integration Patterns Camel supports various Languages to create an Expression or Predicate within either the Routing Domain Specific Language or the XML Configuration. The following languages are supported

BEAN LANGUAGE

The purpose of the Bean Language is to be able to implement an Expression or Predicate using a simple method on a bean. The bean name is resolved using a Registry, such as the Spring ApplicationContext, then a method is invoked to evaluate the Expression or Predicate. If no method name is provided then one is chosen using the rules for Bean Binding; using the type of the message body and using any annotations on the bean methods.

The Bean Binding rules are used to bind the Message Exchange to the method parameters; so you can annotate the bean to extract headers or other expressions such as XPath or XQuery from the message.

Using Bean Expressions in Java

```
from("activemq:topic:OrdersTopic")
    .filter().method("myBean", "isGoldCustomer")
    .to("activemq:BigSpendersQueue");
```

Using Bean Expressions in Spring XML

```
<route>
    <from uri="activemq:topic:OrdersTopic"/>
    <filter>
        <method ref="myBean" method="isGoldCustomer"/>
        <to uri="activemq:BigSpendersQueue"/>
    </filter>
</route>
```

Bean Attribute Now Deprecated

The bean attribute of the method expression element is now deprecated. Use the ref attribute instead.

Writing the Expression Bean

The bean in the above examples is just any old Java Bean with a method called `isGoldCustomer()` that returns some object that is easily converted to a `boolean` value in this case, as its used as a predicate.

Example:

```
public class MyBean {
    public boolean isGoldCustomer(Exchange exchange) {
        // ...
    }
}
```

We can also use the Bean Integration annotations.

Example:

```
public boolean isGoldCustomer(String body) {...}
```

or

```
public boolean isGoldCustomer(@Header(name = "foo") Integer fooHeader) {...}
```

So you can bind parameters of the method to the Exchange, the Message or individual headers, properties, the body or other expressions.

Non-Registry Beans

The Bean Language also supports invoking beans that isn't registered in the Registry. This is usable for quickly to invoke a bean from Java DSL where you don't need to register the bean in the Registry such as the Spring ApplicationContext. Camel can

instantiate the bean and invoke the method if given a class or invoke an already existing instance.

Example:

```
from("activemq:topic:OrdersTopic")
.filter().expression(BeanLanguage.MyBean.class, "isGoldCustomer")
.to("activemq:BigSpendersQueue");
```

The 2nd parameter `isGoldCustomer` is an optional parameter to explicit set the method name to invoke. If not provided Camel will try to invoke the most suitable method. If case of ambiguity Camel will throw an Exception. In these situations the 2nd parameter can solve this problem. Also the code is more readable if the method name is provided. The 1st parameter can also be an existing instance of a Bean such as:

```
private MyBean my;
from("activemq:topic:OrdersTopic")
.filter().expression(BeanLanguage.bean(my, "isGoldCustomer"))
.to("activemq:BigSpendersQueue");
```

In **Camel 2.2**: you can avoid the `BeanLanguage` and have it just as:

```
private MyBean my;
from("activemq:topic:OrdersTopic")
.filter().expression(bean(my, "isGoldCustomer"))
.to("activemq:BigSpendersQueue");
```

Which also can be done in a bit shorter and nice way:

```
private MyBean my;
from("activemq:topic:OrdersTopic")
.filter().method(my, "isGoldCustomer")
.to("activemq:BigSpendersQueue");
```

Other Examples

We have some test cases you can look at if it'll help

- `MethodFilterTest` is a JUnit test case showing the Java DSL use of the bean expression being used in a filter
- `aggregator.xml` is a Spring XML test case for the Aggregator which uses a bean method call to test for the completion of the aggregation.

Dependencies

The Bean language is part of `camel-core`.

CONSTANT EXPRESSION LANGUAGE

The Constant Expression Language is really just a way to specify constant strings as a type of expression.

Example usage

The `setHeader` element of the Spring DSL can utilize a constant expression like:

```
<route>
<from uri="seda:a"/>
<setHeader headerName="theHeader">
<constant>the value</constant>
</setHeader>
<to uri="mock:b"/>
</route>
```

In this case, the Message coming from the `seda:a` Endpoint will have the `theHeader` header set to the constant value `the value`.

And the same example using Java DSL:

```
from("seda:a")
.setHeader("theHeader", constant("the value"))
.to("mock:b");
```

Dependencies

The Constant language is part of `camel-core`.

EL

Camel supports the unified JSP and JSF Expression Language via the JUEL to allow an Expression or Predicate to be used in the DSL or XML Configuration.

For example you could use EL inside a Message Filter in XML

```
<route>
  <from uri="seda:foo"/>
  <filter>
    <el>${in.headers.foo == 'bar'}/</el>
  </filter>
</route>
```

You could also use slightly different syntax, e.g. if the header name is not a valid identifier:

```
<route>
  <from uri="seda:foo"/>
  <filter>
    <el>${in.headers['My Header'] == 'bar'}/</el>
  </filter>
</route>
```

You could use EL to create an Predicate in a Message Filter or as an Expression for a Recipient List

Variables

Variable	Type	Description
exchange	Exchange	the Exchange object
in	Message	the exchange.in message
out	Message	the exchange.out message

Samples

You can use EL dot notation to invoke operations. If you for instance have a body that contains a POJO that has a `getFamilyName` method then you can construct the syntax as follows:

```
"${in.body.familyName}"
```

Dependencies

To use EL in your camel routes you need to add the dependency on `camel-juel` which implements the EL language.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-juel</artifactId>
  <version>x.x.x</version>
</dependency>
```

Otherwise you'll also need to include JUEL.

HEADER EXPRESSION LANGUAGE

The Header Expression Language allows you to extract values of named headers.

Example usage

The recipientList element of the Spring DSL can utilize a header expression like:

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/recipientListWithStringDelimitedHeader.xml}
In this case, the list of recipients are contained in the header 'myHeader'.
```

And the same example in Java DSL:

```
{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/RecipientListWithStringDelimitedHeaderTest.java}
```

And with a slightly different syntax where you use the builder to the fullest (i.e. avoid using parameters but using stacked operations, notice that header is not a parameter but a stacked method call)

```
java from("direct:a").recipientList().header("myHeader");
```

Dependencies

The Header language is part of **camel-core**.

JXPATH

Camel supports JXPath to allow XPath expressions to be used on beans in an Expression or Predicate to be used in the DSL or XML Configuration. For example you could use JXPath to create an Predicate in a Message Filter or as an Expression for a Recipient List.

You can use XPath expressions directly using smart completion in your IDE as follows
`from("queue:foo").filter().xpath("/in/body/foo").to("queue:bar")`

Variables

Variable	Type	Description
this	Exchange	the Exchange object
in	Message	the exchange.in message
out	Message	the exchange.out message

Options

Option	Type	Description
lenient	boolean	Camel 2.11/2.10.5: Allows to turn lenient on the JXPathContext. When turned on this allows the JXPath expression to evaluate against expressions and message bodies which may be invalid / missing data. See more details at the JXPath Documentation This option is by default false.

Using XML configuration

If you prefer to configure your routes in your Spring XML file then you can use JXPath expressions as follows

```
xml <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=" http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
beans.xsd http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd"> <camelContext
id="camel" xmlns="http://activemq.apache.org/camel/schema/spring"> <route> <from uri="activemq:MyQueue"/> <filter>
<jxpath><in/body/name = 'James'</xpath> <to uri="mqseries:SomeOtherQueue"/> </filter> </route> </camelContext> </beans>
```

Examples

Here is a simple example using a JXPath expression as a predicate in a Message Filter

```
{snippet:id=example|lang=java|url=camel/trunk/components/camel-
xpath/src/test/java/org/apache/camel/language/xpath/JXPathFilterTest.java}
```

JXPATH INJECTION

You can use Bean Integration to invoke a method on a bean and use various languages such as JXPath to extract a value from the message and bind it to a method parameter.

For example

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

Loading script from external resource

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as "classpath:", "file:", or "http:". This is done using the following syntax: "resource:scheme:location", eg to refer to a file on the classpath you can do:
`.setHeader("myHeader").xpath("resource:classpath:myxpath.txt")`

Dependencies

To use JXPath in your camel routes you need to add the dependency on **camel-jxpath** which implements the JXPath language.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-jxpath</artifactId> <version>x.x.x</version>
</dependency>
```

Otherwise, you'll also need Commons JXPath.

MVEL

Camel allows Mvel to be used as an Expression or Predicate the DSL or Xml Configuration.

You could use Mvel to create an Predicate in a Message Filter or as an Expression for a Recipient List

You can use Mvel dot notation to invoke operations. If you for instance have a body that contains a POJO that has a `getFamilyName` method then you can construct the syntax as follows:

```
"request.body.familyName"
// or
"getRequest().getBody().getFamilyName()"
```

Variables

Variable	Type	Description
this	Exchange	the Exchange is the root object
exchange	Exchange	the Exchange object
exception	Throwable	the Exchange exception (if any)
exchangeId	String	the exchange id
fault	Message	the Fault message (if any)
request	Message	the exchange.in message
response	Message	the exchange.out message (if any)
properties	Map	the exchange properties
property(name)	Object	the property by the given name
property(name, type)	Type	the property by the given name as the given type

Samples

For example you could use Mvel inside a Message Filter in XML

```
<route>
<from uri="seda:foo"/>
<filter>
  <mvel>request.headers.foo == 'bar'</mvel>
  <to uri="seda:bar"/>
</filter>
</route>
```

And the sample using Java DSL:

```
from("seda:foo").filter().mvel("request.headers.foo == 'bar'").to("seda:bar");
```

Loading script from external resource

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as "classpath:", "file:", or "http:". This is done using the following syntax: "resource:scheme:location", eg to refer to a file on the classpath you can do:

```
.setHeader("myHeader").mvel("resource:classpath:script.mvel")
```

Dependencies

To use Mvel in your camel routes you need to add the a dependency on **camel-mvel** which implements the Mvel language.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-mvel</artifactId>
<version>x.x.x</version>
</dependency>
```

OGNL

Camel allows OGNL to be used as an Expression or Predicate the DSL or XML Configuration.

You could use OGNL to create an Predicate in a Message Filter or as an Expression for a Recipient List

You can use OGNL dot notation to invoke operations. If you for instance have a body that contains a POJO that has a `getFamilyName` method then you can construct the syntax as follows:

```
"request_body.familyName"
// or
"getRequest().getBody().getFamilyName()"
```

Variables

Variable	Type	Description
this	Exchange	the Exchange is the root object
exchange	Exchange	the Exchange object
exception	Throwable	the Exchange exception (if any)
exchangeId	String	the exchange id
fault	Message	the Fault message (if any)
request	Message	the exchange.in message
response	Message	the exchange.out message (if any)
properties	Map	the exchange properties
property(name)	Object	the property by the given name
property(name, type)	Type	the property by the given name as the given type

Samples

For example you could use OGNL inside a Message Filter in XML

```
<route>
  <from uri="seda:foo"/>
  <filter>
    <ognl>request.headers.foo == &#39;bar&#39;</ognl>
    <to uri="seda:bar"/>
  </filter>
</route>
```

And the sample using Java DSL:

```
from("seda:foo").filter().ognl("request.headers.foo == &#39;bar&#39;").to("seda:bar");
```

Loading script from external resource

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as "classpath:", "file:", or "http:". This is done using the following syntax: "resource:scheme:location", eg to refer to a file on the classpath you can do:

```
.setHeader("myHeader").ognl("resource:classpath:mognl.txt")
```

Dependencies

To use OGNL in your camel routes you need to add the a dependency on `camel-ognl` which implements the OGNL language.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ognl</artifactId>
  <version>x.x.x</version>
</dependency>
```

Otherwise, you'll also need OGNL

PROPERTY EXPRESSION LANGUAGE

The Property Expression Language allows you to extract values of named exchange properties.

From **Camel 2.15** onwards the property language has been renamed to exchangeProperty to avoid ambiguity, confusion and clash with properties as a general term. So use exchangeProperty instead of property when using Camel 2.15 onwards.

Example usage

The recipientList element of the Spring DSL can utilize a property expression like:

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/recipientListWithStringDelimitedProperty.xml}
```

In this case, the list of recipients are contained in the property 'myProperty'.

And the same example in Java DSL:

```
{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/RecipientListWithStringDelimitedPropertyTest.java}
```

And with a slightly different syntax where you use the builder to the fullest (i.e. avoid using parameters but using stacked operations, notice that property is not a parameter but a stacked method call)

```
java from("direct:a").recipientList().property("myProperty");
```

Dependencies

The Property language is part of **camel-core**.

SCRIPTING LANGUAGES

Camel supports a number of scripting languages which can be used to create an Expression or Predicate via the standard JSR 223 which is a standard part of Java 6.

The following scripting languages are integrated into the DSL:

Language	DSL keyword
EL	e1
Groovy	groovy
JavaScript	javascript
JSQSL	sql
JXPath	jxpath
MVEL	mvel
OGNL	ognl
PHP	php
Python	python
Ruby	ruby
XPath	xpath
XQuery	xquery

However any JSR 223 scripting language can be used using the generic DSL methods.

ScriptContext Options

The JSR-223 scripting language's scriptContext is pre-configured with the following attributes all set at **ENGINE_SCOPE**.

Attribute	Type	Value
camelContext	org.apache.camel.CamelContext	The Camel Context.
context	org.apache.camel.CamelContext	The Camel Context (cannot be used in groovy).
exchange	org.apache.camel.Exchange	The current Exchange.
properties	org.apache.camel.builder.script.PropertiesFunction	Camel 2.9: Function with a resolve method to make it easier to use Camels Properties component from scripts. See further below for example.
request	org.apache.camel.Message	The IN message.
response	org.apache.camel.Message	Deprecated: The OUT message. The OUT message is null by default. Use the IN message instead.

See Scripting Languages for the list of languages with explicit DSL support.

Passing Additional Arguments to the ScriptingEngine

Available from Camel 2.8

You can provide additional arguments to the `scriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.

```
Example:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-script/src/test/java/org/apache/camel/builder/script/JavaScriptExpressionTest.java}
```

Using Properties Function

Available from Camel 2.9

If you need to use the Properties component from a script to lookup property placeholders, then its a bit cumbersome to do so. For example, to set a header name `myHeader` with a value from a property placeholder, whose key is taken from a header named `foo`.

```
.setHeader("myHeader").groovy("context.resolvePropertyPlaceholders(\"{\" + request.headers.get('foo') + '}\")")  
From Camel 2.9: you can now use the properties function and the same example is simpler:  
java.setHeader("myHeader").groovy("properties.resolve(request.headers.get('foo'))")
```

Loading Script From External Resource

Available from Camel 2.11

You can externalize the script and have Camel load it from a resource such as `classpath:`, `file:`, or `http:`. This is done using the following syntax: `resource:scheme:location` e.g. to refer to a file on the classpath you can do:

```
java.setHeader("myHeader").groovy("resource:classpath:mygroovy.groovy")
```

How to Get the Result from Multiple Statements Script

Available from Camel 2.14

The script engine's `eval` method returns a `null` when it runs a multi-statement script. However, Camel can look up the value of a script's result by using the key `result` from the value set. When writing a multi-statement script set the value of the `result` variable as the script return value.

```
textbar = "baz"; # some other statements ... # camel take the result value as the script evaluation result result = body * 2 + 1
```

Dependencies

To use scripting languages in your camel routes you need to add the a dependency on `camel-script` which integrates the JSR-223 scripting engine.

If you use maven you could just add the following to your `pom.xml`, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-script</artifactId> <version>x.x.x</version> </dependency>
```

SEE ALSO

- Languages
- DSL
- Xml Configuration

BEANSHELL

Camel supports BeanShell among other Scripting Languages to allow an Expression or Predicate to be used in the DSL or Xml Configuration.

To use a BeanShell expression use the following Java code:

```
...choice()  
  .when(script("beanshell", "request.getHeaders().get(\"foo\").equals(\"bar\")"))  
  .to("...")
```

Or the something like this in your Spring XML:

```
<filter>  
  <language language="beanshell">request.getHeaders().get("Foo") == null</language>  
  ...
```

BeanShell Issues

You must use BeanShell 2.0b5 or greater. Note that as of 2.0b5 BeanShell cannot compile scripts, which causes Camel releases before 2.6 to fail when configured with BeanShell expressions.

You could follow the examples above to create an Predicate in a Message Filter or as an Expression for a Recipient List

ScriptContext Options

The JSR-223 scripting language's `scriptContext` is pre-configured with the following attributes all set at `ENGINE_SCOPE`.

Attribute	Type	Value
<code>camelContext</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context.
<code>context</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context (cannot be used in groovy).
<code>exchange</code>	<code>org.apache.camel.Exchange</code>	The current Exchange.
<code>properties</code>	<code>org.apache.camel.builder.script.PropertiesFunction</code>	Camel 2.9: Function with a <code>resolve</code> method to make it easier to use Camels Properties component from scripts. See further below for example.
<code>request</code>	<code>org.apache.camel.Message</code>	The IN message.
<code>response</code>	<code>org.apache.camel.Message</code>	Deprecated: The OUT message. The OUT message is null by default. Use the IN message instead.

See Scripting Languages for the list of languages with explicit DSL support.

Passing Additional Arguments to the ScriptingEngine

Available from Camel 2.8

You can provide additional arguments to the `ScriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.

```
Example:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-script/src/test/java/org/apache/camel/builder/script/JavaScriptExpressionTest.java}
```

Using Properties Function

Available from Camel 2.9

If you need to use the Properties component from a script to lookup property placeholders, then its a bit cumbersome to do so. For example, to set a header name `myHeader` with a value from a property placeholder, whose key is taken from a header named `foo`.

```
.setHeader("myHeader").groovy("context.resolvePropertyPlaceholders('{" + request.headers.get('foo') + '}')")
```

From **Camel 2.9:** you can now use the properties function and the same example is simpler:

```
java.setHeader("myHeader").groovy("properties.resolve(request.headers.get('foo'))")
```

Loading Script From External Resource

Available from Camel 2.11

You can externalize the script and have Camel load it from a resource such as `classpath:`, `file:`, or `http:`. This is done using the following syntax: `resource:scheme:location` e.g. to refer to a file on the classpath you can do:

```
java.setHeader("myHeader").groovy("resource:classpath:mygroovy.groovy")
```

How to Get the Result from Multiple Statements Script

Available from Camel 2.14

The script engine's `eval` method returns a `null` when it runs a multi-statement script. However, Camel can look up the value of a script's result by using the key `result` from the value set. When writing a multi-statement script set the value of the `result` variable as the script return value.

```
textbar = "baz"; # some other statements ... # camel take the result value as the script evaluation result result = body * 2 + 1
```

Dependencies

To use scripting languages in your camel routes you need to add the dependency on `camel-script` which integrates the JSR-223 scripting engine.

If you use maven you could just add the following to your `pom.xml`, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-script</artifactId> <version>x.x.x</version>
</dependency>
```

JAVASCRIPT

Camel supports JavaScript/ECMAScript among other Scripting Languages to allow an Expression or Predicate to be used in the DSL or Xml Configuration.

To use a JavaScript expression use the following Java code

```
... javaScript("someJavaScriptExpression") ...
```

For example you could use the **javaScript** function to create an Predicate in a Message Filter or as an Expression for a Recipient List

Example

In the sample below we use JavaScript to create a Predicate use in the route path, to route exchanges from admin users to a special queue.

```
from("direct:start")
    .choice()
        .when().javaScript("request.headers.get('user') == 'admin'")
        .otherwise()
            .to("seda:regularQueue");
```

And a Spring DSL sample as well:

```
<route>
    <from uri="direct:start"/>
    <choice>
        <when>
            <javaScript>request.headers.get('user') == 'admin'</javaScript>
            <to uri="seda:adminQueue"/>
        </when>
        <otherwise>
            <to uri="seda:regularQueue"/>
        </otherwise>
    </choice>
</route>
```

ScriptContext Options

The JSR-223 scripting language's `scriptContext` is pre-configured with the following attributes all set at `ENGINE_SCOPE`.

Attribute	Type	Value
<code>camelContext</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context.
<code>context</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context (cannot be used in groovy).
<code>exchange</code>	<code>org.apache.camel.Exchange</code>	The current Exchange.
<code>properties</code>	<code>org.apache.camel.builder.script.PropertiesFunction</code>	Camel 2.9: Function with a <code>resolve</code> method to make it easier to use Camels Properties component from scripts. See further below for example.
<code>request</code>	<code>org.apache.camel.Message</code>	The <code>IN</code> message.
<code>response</code>	<code>org.apache.camel.Message</code>	Deprecated: The <code>out</code> message. The <code>out</code> message is <code>null</code> by default. Use the <code>IN</code> message instead.

See Scripting Languages for the list of languages with explicit DSL support.

Passing Additional Arguments to the ScriptingEngine

Available from Camel 2.8

You can provide additional arguments to the `scriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.

Example:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-script/src/test/java/org/apache/camel/builder/script/JavaScriptExpressionTest.java}

Using Properties Function

Available from Camel 2.9

If you need to use the Properties component from a script to lookup property placeholders, then its a bit cumbersome to do so. For example, to set a header name `myHeader` with a value from a property placeholder, whose key is taken from a header named `foo`.

```
.setHeader("myHeader").groovy("context.resolvePropertyPlaceholders('{{' + request.headers.get('foo') + '}}')")
```

From **Camel 2.9**: you can now use the properties function and the same example is simpler:

```
java.setHeader("myHeader").groovy("properties.resolve(request.headers.get('foo'))")
```

Loading Script From External Resource

Available from Camel 2.11

You can externalize the script and have Camel load it from a resource such as `classpath:`, `file:`, or `http:`. This is done using the following syntax: `resource:scheme:location` e.g. to refer to a file on the classpath you can do:

```
java.setHeader("myHeader").groovy("resource:classpath:mygroovy.groovy")
```

How to Get the Result from Multiple Statements Script

Available from Camel 2.14

The script engine's eval method returns a null when it runs a multi-statement script. However, Camel can look up the value of a script's result by using the key `result` from the value set. When writing a multi-statement script set the value of the `result` variable as the script return value.

```
textbar = "baz"; # some other statements ... # camel take the result value as the script evaluation result result = body * 2 + 1
```

Dependencies

To use scripting languages in your camel routes you need to add the a dependency on `camel-script` which integrates the JSR-223 scripting engine.

If you use maven you could just add the following to your `pom.xml`, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-script</artifactId> <version>x.x.x</version>
</dependency>
```

GROOVY

Camel supports Groovy among other Scripting Languages to allow an Expression or Predicate to be used in the DSL or XML Configuration.

To use a Groovy expression use the following Java code

```
... groovy("someGroovyExpression") ...
```

For example you could use the `groovy` function to create an Predicate in a Message Filter or as an Expression for a Recipient List

Dependency

You should add the `camel-groovy` dependency when using Groovy language with Camel. The generic `camel-script` is not optimized for best Groovy experience, and hence you should add `camel-groovy` as dependency.

Customizing Groovy Shell

Sometimes you may need to use custom `GroovyShell` instance in your Groovy expressions. To provide custom `GroovyShell`, add implementation of the `org.apache.camel.language.groovy.GroovyShellFactory` SPI interface to your Camel registry. For example after adding the following bean to your Spring context...

```
public class CustomGroovyShellFactory implements GroovyShellFactory {
    public GroovyShell createGroovyShell(Exchange exchange) {
        ImportCustomizer importCustomizer = new ImportCustomizer();
        importCustomizer.addStaticStars("com.example.Utils");
        CompilerConfiguration configuration = new CompilerConfiguration();
        configuration.addCompilationCustomizers(importCustomizer);
        return new GroovyShell(configuration);
    }
}
```

...Camel will use your custom `GroovyShell` instance (containing your custom static imports), instead of the default one.

Example

```
// lets route if a line item is over $100
from("queue:foo").filter(groovy("request.lineItems.any { i -> i.value > 100 }")).to("queue:bar")
```

And the Spring DSL:

```
<route>
  <from uri="queue:foo"/>
  <filter>
    <groovy>request.lineItems.any { i -> i.value > 100 }</groovy>
  <to uri="queue:bar"/>
</filter>
</route>
```

ScriptContext Options

The JSR-223 scripting language's `scriptContext` is pre-configured with the following attributes all set at `ENGINE_SCOPE`.

Attribute	Type	Value
<code>camelContext</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context.
<code>context</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context (cannot be used in groovy).
<code>exchange</code>	<code>org.apache.camel.Exchange</code>	The current Exchange.
<code>properties</code>	<code>org.apache.camel.builder.script.PropertiesFunction</code>	Camel 2.9: Function with a <code>resolve</code> method to make it easier to use Camels Properties component from scripts. See further below for example.
<code>request</code>	<code>org.apache.camel.Message</code>	The <code>IN</code> message.
<code>response</code>	<code>org.apache.camel.Message</code>	Deprecated: The <code>OUT</code> message. The <code>OUT</code> message is <code>null</code> by default. Use the <code>IN</code> message instead.

See Scripting Languages for the list of languages with explicit DSL support.

Passing Additional Arguments to the ScriptingEngine

Available from Camel 2.8

You can provide additional arguments to the `ScriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.

Example:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-script/src/test/java/org/apache/camel/builder/script/JavaScriptExpressionTest.java}

Using Properties Function

Available from Camel 2.9

If you need to use the Properties component from a script to lookup property placeholders, then its a bit cumbersome to do so. For example, to set a header name `myHeader` with a value from a property placeholder, whose key is taken from a header named `foo`.

`.setHeader("myHeader").groovy("context.resolvePropertyPlaceholders('{{' + request.headers.get('foo') + '}}')")`

From **Camel 2.9:** you can now use the properties function and the same example is simpler:

`java.setHeader("myHeader").groovy("properties.resolve(request.headers.get('foo'))")`

Loading Script From External Resource

Available from Camel 2.11

You can externalize the script and have Camel load it from a resource such as `classpath:`, `file:`, or `http:`. This is done using the following syntax: `resource:scheme:location` e.g. to refer to a file on the classpath you can do:

`java.setHeader("myHeader").groovy("resource:classpath:mygroovy.groovy")`

How to Get the Result from Multiple Statements Script

Available from Camel 2.14

The script engine's `eval` method returns a `null` when it runs a multi-statement script. However, Camel can look up the value of a script's result by using the key `result` from the value set. When writing a multi-statement script set the value of the `result` variable as the script return value.

`textbar = "baz"; # some other statements ... # camel take the result value as the script evaluation result result = body * 2 + 1`

Dependencies

To use scripting languages in your camel routes you need to add the a dependency on `camel-script` which integrates the JSR-223 scripting engine.

If you use maven you could just add the following to your `pom.xml`, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-script</artifactId> <version>x.x.x</version>
</dependency>
```

PYTHON

Camel supports Python among other Scripting Languages to allow an Expression or Predicate to be used in the DSL or XML Configuration.

To use a Python expression use the following Java code

```
... python("somePythonExpression") ...
```

For example you could use the `python` function to create an Predicate in a Message Filter or as an Expression for a Recipient List

Example

In the sample below we use Python to create a Predicate use in the route path, to route exchanges from admin users to a special queue.

```
from("direct:start")
    .choice()
        .when().python("request.headers[user] == 'admin'")
        .otherwise()
            .to("seda:regularQueue");
```

And a Spring DSL sample as well:

```
<route>
    <from uri="direct:start"/>
    <choice>
        <when>
            <python>request.headers[user] == 'admin'</python>
            <to uri="seda:adminQueue"/>
        </when>
        <otherwise>
            <to uri="seda:regularQueue"/>
        </otherwise>
    </choice>
</route>
```

ScriptContext Options

The JSR-223 scripting language's `scriptContext` is pre-configured with the following attributes all set at `ENGINE_SCOPE`.

Attribute	Type	Value
<code>camelContext</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context.
<code>context</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context (cannot be used in groovy).
<code>exchange</code>	<code>org.apache.camel.Exchange</code>	The current Exchange.
<code>properties</code>	<code>org.apache.camel.builder.script.PropertiesFunction</code>	Camel 2.9: Function with a <code>resolve</code> method to make it easier to use Camels Properties component from scripts. See further below for example.
<code>request</code>	<code>org.apache.camel.Message</code>	The IN message.
<code>response</code>	<code>org.apache.camel.Message</code>	Deprecated: The OUT message. The out message is <code>null</code> by default. Use the IN message instead.

See Scripting Languages for the list of languages with explicit DSL support.

Passing Additional Arguments to the ScriptingEngine

Available from Camel 2.8

You can provide additional arguments to the ScriptingEngine using a header on the Camel message with the key `CamelScriptArguments`.

```
Example:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-
script/src/test/java/org/apache/camel/builder/script/JavaScriptExpressionTest.java}
```

Using Properties Function

Available from Camel 2.9

If you need to use the Properties component from a script to lookup property placeholders, then its a bit cumbersome to do so. For example, to set a header name `myHeader` with a value from a property placeholder, whose key is taken from a header named `foo`.

```
.setHeader("myHeader").groovy("context.resolvePropertyPlaceholders('{" + request.headers.get('foo') + '}')")
```

From **Camel 2.9**: you can now use the properties function and the same example is simpler:

```
java.setHeader("myHeader").groovy("properties.resolve(request.headers.get('foo'))")
```

Loading Script From External Resource

Available from Camel 2.11

You can externalize the script and have Camel load it from a resource such as `classpath:`, `file:`, or `http:`. This is done using the following syntax: `resource:scheme:location` e.g. to refer to a file on the classpath you can do:

```
java.setHeader("myHeader").groovy("resource:classpath:mygroovy.groovy")
```

How to Get the Result from Multiple Statements Script

Available from Camel 2.14

The script engine's eval method returns a null when it runs a multi-statement script. However, Camel can look up the value of a script's result by using the key `result` from the value set. When writing a multi-statement script set the value of the `result` variable as the script return value.

```
textbar = "baz"; # some other statements ... # camel take the result value as the script evaluation result result = body * 2 + 1
```

Dependencies

To use scripting languages in your camel routes you need to add the dependency on `camel-script` which integrates the JSR-223 scripting engine.

If you use maven you could just add the following to your `pom.xml`, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-script</artifactId> <version>x.x.x</version>
</dependency>
```

PHP

Camel supports PHP among other Scripting Languages to allow an Expression or Predicate to be used in the DSL or XML Configuration.

To use a PHP expression use the following Java code

```
... php("somePHPExpression") ...
```

For example you could use the `php` function to create an Predicate in a Message Filter or as an Expression for a Recipient List

ScriptContext Options

The JSR-223 scripting language's `scriptContext` is pre-configured with the following attributes all set at `ENGINE_SCOPE`.

Attribute	Type	Value
<code>camelContext</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context.
<code>context</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context (cannot be used in groovy).
<code>exchange</code>	<code>org.apache.camel.Exchange</code>	The current Exchange.
<code>properties</code>	<code>org.apache.camel.builder.script.PropertiesFunction</code>	Camel 2.9: Function with a <code>resolve</code> method to make it easier to use Camels Properties component from scripts. See further below for example.

request	org.apache.camel.Message	The IN message.
response	org.apache.camel.Message	Deprecated: The OUT message. The OUT message is null by default. Use the IN message instead.

See Scripting Languages for the list of languages with explicit DSL support.

Passing Additional Arguments to the ScriptingEngine

Available from Camel 2.8

You can provide additional arguments to the ScriptingEngine using a header on the Camel message with the key CamelScriptArguments.

```
Example:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-script/src/test/java/org/apache/camel/builder/script/JavaScriptExpressionTest.java}
```

Using Properties Function

Available from Camel 2.9

If you need to use the Properties component from a script to lookup property placeholders, then its a bit cumbersome to do so. For example, to set a header name myHeader with a value from a property placeholder, whose key is taken from a header named foo.

```
.setHeader("myHeader").groovy("context.resolvePropertyPlaceholders(\"{\" + request.headers.get('foo') + '}\"")")
```

From Camel 2.9: you can now use the properties function and the same example is simpler:

```
java.setHeader("myHeader").groovy("properties.resolve(request.headers.get('foo'))")
```

Loading Script From External Resource

Available from Camel 2.11

You can externalize the script and have Camel load it from a resource such as classpath:, file:, or http:. This is done using the following syntax: resource:scheme:location e.g. to refer to a file on the classpath you can do:

```
java.setHeader("myHeader").groovy("resource:classpath:mygroovy.groovy")
```

How to Get the Result from Multiple Statements Script

Available from Camel 2.14

The script engine's eval method returns a null when it runs a multi-statement script. However, Camel can look up the value of a script's result by using the key result from the value set. When writing a multi-statement script set the value of the result variable as the script return value.

```
textbar = "baz"; # some other statements ... # camel take the result value as the script evaluation result result = body * 2 + 1
```

Dependencies

To use scripting languages in your camel routes you need to add the a dependency on camel-script which integrates the JSR-223 scripting engine.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-script</artifactId> <version>x.x.x</version>
</dependency>
```

RUBY

Camel supports Ruby among other Scripting Languages to allow an Expression or Predicate to be used in the DSL or XML Configuration.

To use a Ruby expression use the following Java code

```
... ruby("someRubyExpression") ...
```

For example you could use the ruby function to create an Predicate in a Message Filter or as an Expression for a Recipient List

Example

In the sample below we use Ruby to create a Predicate use in the route path, to route exchanges from admin users to a special queue.

```
from("direct:start")
    .choice()
        .when().ruby("$.request.headers[user] == 'admin').to("seda:adminQueue")
```

```
.otherwise()
.to("seda:regularQueue");
```

And a Spring DSL sample as well:

```
<route>
  <from uri="direct:start"/>
  <choice>
    <when>
      <ruby>$request.headers['user'] == 'admin'</ruby>
      <to uri="seda:adminQueue"/>
    </when>
    <otherwise>
      <to uri="seda:regularQueue"/>
    </otherwise>
  </choice>
</route>
```

ScriptContext Options

The JSR-223 scripting language's `scriptContext` is pre-configured with the following attributes all set at `ENGINE_SCOPE`.

Attribute	Type	Value
<code>camelContext</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context.
<code>context</code>	<code>org.apache.camel.CamelContext</code>	The Camel Context (cannot be used in groovy).
<code>exchange</code>	<code>org.apache.camel.Exchange</code>	The current Exchange.
<code>properties</code>	<code>org.apache.camel.builder.script.PropertiesFunction</code>	Camel 2.9: Function with a <code>resolve</code> method to make it easier to use Camels Properties component from scripts. See further below for example.
<code>request</code>	<code>org.apache.camel.Message</code>	The <code>IN</code> message.
<code>response</code>	<code>org.apache.camel.Message</code>	Deprecated: The <code>OUT</code> message. The <code>OUT</code> message is <code>null</code> by default. Use the <code>IN</code> message instead.

See Scripting Languages for the list of languages with explicit DSL support.

Passing Additional Arguments to the ScriptingEngine

Available from Camel 2.8

You can provide additional arguments to the `ScriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.

Example:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-script/src/test/java/org/apache/camel/builder/script/JavaScriptExpressionTest.java}

Using Properties Function

Available from Camel 2.9

If you need to use the Properties component from a script to lookup property placeholders, then its a bit cumbersome to do so. For example, to set a header name `myHeader` with a value from a property placeholder, whose key is taken from a header named `foo`.

`.setHeader("myHeader").groovy("context.resolvePropertyPlaceholders('{" + request.headers.get('foo') + '}')")`

From **Camel 2.9:** you can now use the properties function and the same example is simpler:

`java.setHeader("myHeader").groovy("properties.resolve(request.headers.get('foo'))")`

Loading Script From External Resource

Available from Camel 2.11

You can externalize the script and have Camel load it from a resource such as `classpath:`, `file:`, or `http:`. This is done using the following syntax: `resource:scheme:location` e.g. to refer to a file on the classpath you can do:

`java.setHeader("myHeader").groovy("resource:classpath:mygroovy.groovy")`

How to Get the Result from Multiple Statements Script

Available from Camel 2.14

The script engine's `eval` method returns a `null` when it runs a multi-statement script. However, Camel can look up the value of a script's result by using the key `result` from the value set. When writing a multi-statement script set the value of the `result` variable as the script return value.

`textbar = "baz"; # some other statements ... # camel take the result value as the script evaluation result result = body * 2 + 1`

Dependencies

To use scripting languages in your camel routes you need to add the a dependency on `camel-script` which integrates the JSR-223 scripting engine.

If you use maven you could just add the following to your `pom.xml`, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-script</artifactId> <version>x.x.x</version>
</dependency>
```

SIMPLE EXPRESSION LANGUAGE

The Simple Expression Language was a really simple language when it was created, but has since grown more powerful. It is primarily intended for being a really small and simple language for evaluating Expressions and Predicates without requiring any new dependencies or knowledge of XPath; so it is ideal for testing in `camel-core`. The idea was to cover 95% of the common use cases when you need a little bit of expression based script in your Camel routes.

However for much more complex use cases you are generally recommended to choose a more expressive and powerful language such as:

- SpEL
- Mvel
- Groovy
- JavaScript
- EL
- OGNL
- one of the supported Scripting Languages

The simple language uses `${body}` placeholders for complex expressions where the expression contains constant literals.

Deprecated: The `${}` placeholders can be omitted if the expression starts with the token, or if the token is only itself.

Alternative syntax

From Camel 2.5 you can also use the alternative syntax which uses `$simple{}` as placeholders. This can be used in situations to avoid clashes when using for example Spring property placeholder together with Camel.

Configuring result type

From Camel 2.8 you can configure the result type of the Simple expression. For example to set the type as a `java.lang.Boolean` or a `java.lang.Integer` etc.

File language is now merged with Simple language

From Camel 2.2, the File Language is now merged with Simple language which means you can use all the file syntax directly within the simple language.

Simple Language Changes in Camel 2.9 onwards

The Simple language have been improved from Camel 2.9 to use a better syntax parser, which can do index precise error messages, so you know exactly what is wrong and where the problem is. For example if you have made a typo in one of the operators, then previously the parser would not be able to detect this, and cause the evaluation to be true. There are a few changes in the syntax which are no longer backwards compatible. When using Simple language as a Predicate then the literal text **must** be enclosed in either single or double quotes. For example: `"${body} == 'Camel'"`. Notice how we have single quotes around the literal. The old style of using "body" and "header.foo" to refer to the message body and header is @deprecated, and it is encouraged to always use `${}` tokens for the built-in functions.

The range operator now requires the range to be in single quote as well as shown: `"${header.zip} between '30000 .. 39999'"`.

To get the body of the in message: body, or `in.body` or `${body}` .

A complex expression must use `${}` placeholders, such as: `Hello ${in.header.name} how are you?`.

You can have multiple functions in the same expression: `"Hello ${in.header.name} this is ${in.header.me} speaking"`. However you can **not** nest functions in Camel 2.8.x or older e.g., having another `${}` placeholder in an existing, is not allowed. From **Camel 2.9** you can nest functions.

Variables

confluenceTableSmall

Variable	Type	Description
<code>camelId</code>	<code>String</code>	Camel 2.10: the CamelContext name.
<code>camelContext.ognl</code>	<code>Object</code>	Camel 2.11: the CamelContext invoked using a Camel OGNL expression.
<code>collate(group)</code>	<code>List</code>	Camel 2.17: The collate function iterates the message body and groups the data into sub lists of specified size. This can be used with the Splitter EIP to split a message body and group/batch the split sub messages into a group of N sub lists. This method works similar to the collate method in Groovy.

<code>exchange</code>	<code>Exchange</code>	Camel 2.16: the Exchange.
<code>exchange.OGNL</code>	<code>Object</code>	Camel 2.16: the Exchange invoked using a Camel OGNL expression.
<code>exchangeId</code>	<code>String</code>	Camel 2.3: the exchange Id.
<code>id</code>	<code>String</code>	The input message Id.
<code>body</code>	<code>Object</code>	The input body.
<code>in.body</code>	<code>Object</code>	The input body.
<code>body.OGNL</code>	<code>Object</code>	Camel 2.3: the input body invoked using a Camel OGNL expression.
<code>in.body.OGNL</code>	<code>Object</code>	Camel 2.3: the input body invoked using a Camel OGNL expression.
<code>bodyAs(type)</code>	<code>Type</code>	Camel 2.3: Converts the body to the given type determined by its classname. The converted body can be null.
<code>bodyAs(type).OGNL</code>	<code>Object</code>	Camel 2.18: Converts the body to the given type determined by its classname and then invoke methods using a Camel OGNL expression. The converted body can be null.
<code>mandatoryBodyAs(type)</code>	<code>Type</code>	Camel 2.5: Converts the body to the given type determined by its classname, and expects the body to be not null.
<code>mandatoryBodyAs(type).OGNL</code>	<code>Object</code>	Camel 2.18: Converts the body to the given type determined by its classname and then invoke methods using a Camel OGNL expression.
<code>out.body</code>	<code>Object</code>	The output body.
<code>header.foo</code>	<code>Object</code>	Refer to the input foo header.
<code>header[foo]</code>	<code>Object</code>	Camel 2.9.2: refer to the input foo header.
<code>headers.foo</code>	<code>Object</code>	Refer to the input foo header.
<code>headers[foo]</code>	<code>Object</code>	Camel 2.9.2: refer to the input foo header.
<code>in.header.foo</code>	<code>Object</code>	Refer to the input foo header.
<code>in.header[foo]</code>	<code>Object</code>	Camel 2.9.2: refer to the input foo header.
<code>in.headers.foo</code>	<code>Object</code>	Refer to the input foo header.
<code>in.headers[foo]</code>	<code>Object</code>	Camel 2.9.2: refer to the input foo header.
<code>header.foo[bar]</code>	<code>Object</code>	Camel 2.3: regard input foo header as a map and perform lookup on the map with bar as key.
<code>in.header.foo[bar]</code>	<code>Object</code>	Camel 2.3: regard input foo header as a map and perform lookup on the map with bar as key.
<code>in.headers.foo[bar]</code>	<code>Object</code>	Camel 2.3: regard input foo header as a map and perform lookup on the map with bar as key.
<code>header.foo.OGNL</code>	<code>Object</code>	Camel 2.3: refer to the input foo header and invoke its value using a Camel OGNL expression.
<code>in.header.foo.OGNL</code>	<code>Object</code>	Camel 2.3: refer to the input foo header and invoke its value using a Camel OGNL expression.
<code>in.headers.foo.OGNL</code>	<code>Object</code>	Camel 2.3: refer to the input foo header and invoke its value using a Camel OGNL expression.
<code>out.header.foo</code>	<code>Object</code>	Refer to the out header foo.
<code>out.header[foo]</code>	<code>Object</code>	Camel 2.9.2: refer to the out header foo.
<code>out.headers.foo</code>	<code>Object</code>	Refer to the out header foo.
<code>out.headers[foo]</code>	<code>Object</code>	Camel 2.9.2: refer to the out header foo.
<code>headerAs(key, type)</code>	<code>Type</code>	Camel 2.5: Converts the header to the given type determined by its classname.
<code>headers</code>	<code>Map</code>	Camel 2.9: refer to the input headers.
<code>in.headers</code>	<code>Map</code>	Camel 2.9: refer to the input headers.
<code>property.foo</code>	<code>Object</code>	Deprecated: refer to the foo property on the exchange.
<code>exchangeProperty.foo</code>	<code>Object</code>	Camel 2.15: refer to the foo property on the exchange.
<code>property[foo]</code>	<code>Object</code>	Deprecated: refer to the foo property on the exchange.
<code>exchangeProperty[foo]</code>	<code>Object</code>	Camel 2.15: refer to the foo property on the exchange.
<code>property.foo.OGNL</code>	<code>Object</code>	Deprecated: refer to the foo property on the exchange and invoke its value using a Camel OGNL expression.
<code>exchangeProperty.foo.OGNL</code>	<code>Object</code>	Camel 2.15: refer to the foo property on the exchange and invoke its value using a Camel OGNL expression.
<code>sys.foo</code>	<code>String</code>	

		Refer to the system property foo.
sysenv.foo	String	Camel 2.3: refer to the system environment property foo.
exception	object	Camel 2.4: Refer to the exception object on the exchange, is null if no exception set on exchange. Will fallback and grab caught exceptions (Exchange.EXCEPTION_CAUGHT) if the Exchange has any.
exception.OGNL	object	Camel 2.4: Refer to the exchange exception invoked using a Camel OGNL expression object
exception.message	String	Refer to the exception.message on the exchange, is null if no exception set on exchange. Will fallback and grab caught exceptions (Exchange.EXCEPTION_CAUGHT) if the Exchange has any.
exception.stacktrace	String	Camel 2.6. Refer to the exception.stacktrace on the exchange. Result is null if no exception set on exchange. Will fallback and grab caught exceptions (Exchange.EXCEPTION_CAUGHT) if the Exchange has any.
		Date formatting using the java.text.SimpleDateFormat patterns. Supported commands are:
date:command:pattern	String	<ul style="list-style-type: none"> • now for current timestamp. • in.header.xxx or header.xxx to use the Date object in the IN header with the key xxx. • out.header.xxx to use the date object in the OUT header with the key xxx.
bean:bean expression	Object	Invoking a bean expression using the Bean language. Specifying a method name you must use dot as separator. We also support the ? method=methodname syntax that is used by the Bean component.
properties:locations:key	String	Deprecated: (use properties-location instead) Camel 2.3: Lookup a property with the given key. The locations option is optional. See more at Using PropertyPlaceholder.
properties-location:locations:key	String	Camel 2.14.1: Lookup a property with the given key. The locations option is optional. See more at Using PropertyPlaceholder.
properties:key:default	String	Camel 2.14.1: Lookup a property with the given key. If the key does not exists or has no value, then an optional default value can be specified.
routeId	String	Camel 2.11: Returns the Id of the current route the Exchange is being routed.
threadName	String	Camel 2.3: Returns the name of the current thread. Can be used for logging purpose.
ref:xxx	Object	Camel 2.6: To lookup a bean from the Registry with the given Id.
type:name.field	Object	Camel 2.11: To refer to a type or field by its FQN name. To refer to a field you can append .FIELD_NAME. For example you can refer to the constant field from Exchange as: org.apache.camel.Exchange.FILE_NAME
null	null	Camel 2.12.3: represents a null.
random(value)	Integer	Camel 2.16.0: returns a random Integer between 0 (included) and value (excluded)
random(min,max)	Integer	Camel 2.16.0: returns a random Integer between min (included) and max (excluded)
skip(number)	Iterator	Camel 2.19: The skip function iterates the message body and skips the first number of items. This can be used with the Splitter EIP to split a message body and skip the first N number of items.
messageHistory	String	Camel 2.17: The message history of the current exchange how it has been routed. This is similar to the route stack-trace message history the error handler logs in case of an unhandled exception.
messageHistory(false)	String	Camel 2.17: As messageHistory but without the exchange details (only includes the route stack-trace). This can be used if you do not want to log sensitive data from the message itself.

OGNL expression support

Available as of Camel 2.3

Camel's OGNL support is for invoking methods only. You cannot access fields.

From **Camel 2.11.1:** we added special support for accessing the length field of Java arrays.

The Simple and Bean language now supports a Camel OGNL notation for invoking beans in a chain like fashion. Suppose the Message IN body contains a POJO which has a getAddress() method.

Then you can use Camel OGNL notation to access the address object:

```
javasimple("${body.address}") simple("${body.address.street}") simple("${body.address.zip}")
Camel understands the shorthand names for accessors, but you can invoke any method or use the real name such as:
javasimple("${body.address}") simple("${body.getAddress.street}") simple("${body.address.getZip}")
simple("${body.doSomething}")
```

You can also use the null safe operator (?) to avoid a NPE if for example the body does *not* have an address
`javasimple("${body?.address?.street}")`

It is also possible to index in Map or List types, so you can do:

```
javasimple("${body[foo].name}")
```

To assume the body is Map based and lookup the value with foo as key, and invoke the getName method on that value.
 Key with spaces

If the key has space, then you **must** enclose the key with quotes, for example:

```
javasimple("${body['foo bar'].name}")
```

You can access the Map or List objects directly using their key name (with or without dots) :

```
javasimple("${body[foo]}") simple("${body[this.is.foo]}")
```

Suppose there was no value with the key foo then you can use the null safe operator to avoid a NPE as shown:

```
javasimple("${body[foo]?.name}")
```

You can also access List types, for example to get lines from the address you can do:

```
javasimple("${body.address.lines[0]}") simple("${body.address.lines[1]}") simple("${body.address.lines[2]}")
```

There is a special last keyword which can be used to get the last value from a list.

```
javasimple("${body.address.lines[last]}")
```

And to get the penultimate line use subtraction. In this case use last-1 for this:

```
javasimple("${body.address.lines[last-1]}")
```

And the third last is of course:

```
javasimple("${body.address.lines[last-2]}")
```

And you can call the size method on the list with

```
javasimple("${body.address.lines.size}")
```

From **Camel 2.11** we added support for the length field for Java arrays as well. Example:

```
javaString[] lines = new String[]{"foo", "bar", "cat"}; exchange.getIn().setBody(lines); simple("There are ${body.length} lines")
```

And yes you can combine this with the operator support as shown below:

```
javasimple("${body.address.zip} > 1000")
```

Operator Support

The parser is limited to only support a single operator. To enable it the left value must be enclosed in \${}.

The syntax is:

```
java${leftValue} OP rightValue
```

Where the rightvalue can be a string literal enclosed in ' ', null, a constant value or another expression enclosed in \${}.
 Important

There **must** be spaces around the operator.

Camel will automatically type convert the rightvalue type to the leftvalue type, so it is possible to for example, convert a string into a numeric so you can use > comparison for numeric values.

The following operators are supported:

Operator	Description
==	Equals.
=~	Camel 2.16: equals ignore case (will ignore case when comparing string values).
>	Greater than.
>=	Greater than or equals.
<	Less than.
<=	Less than or equals.
!=	Not equals.
contains	For testing if contains in a string based value.
not contains	For testing if not contains in a string based value.
regex	For matching against a given regular expression pattern defined as a string value.
not regex	For not matching against a given regular expression pattern defined as a string value.
in	For matching if in a set of values, each element must be separated by comma. If you want to include an empty value, then it must be defined using double comma, eg ',bronze,silver,gold', which is a set of four values with an empty value and then the three medals.
not in	For matching if not in a set of values, each element must be separated by comma. If you want to include an empty value, then it must be defined using double comma. Example: ' ,bronze,silver,gold', which is a set of four values with an empty value and then the three medals.

<code>is</code>	For matching if the left hand side type is an instanceof the value.
<code>not is</code>	For matching if the left hand side type is not an instanceof the value.
<code>range</code>	For matching if the left hand side is within a range of values defined as numbers: from...to. From Camel 2.9: the range values must be enclosed in single quotes.
<code>not range</code>	For matching if the left hand side is not within a range of values defined as numbers: from...to. From Camel 2.9: the range values must be enclosed in single quotes.
<code>starts with</code>	Camel 2.17.1, 2.18: For testing if the left hand side string starts with the right hand string.
<code>ends with</code>	Camel 2.17.1, 2.18: For testing if the left hand side string ends with the right hand string.

And the following unary operators can be used:

Operator	Description
<code>++</code>	Camel 2.9: To increment a number by one. The left hand side must be a function, otherwise parsed as literal.
<code>--</code>	Camel 2.9: To decrement a number by one. The left hand side must be a function, otherwise parsed as literal.
<code>Camel 2.9.3 to 2.10.x</code>	To escape a value, e.g., <code>\\$</code> , to indicate a \$ sign. Special: Use <code>\n</code> for new line, <code>\t</code> for tab, and <code>\r</code> for carriage return.
<code>\</code>	Note: Escaping is not supported using the File Language. Note: from Camel 2.11, the escape character is no longer supported. It has been replaced with the following three escape sequences.
<code>\n</code>	Camel 2.11: To use newline character.
<code>\t</code>	Camel 2.11: To use tab character.
<code>\r</code>	Camel 2.11: To use carriage return character.
<code>\}</code>	Camel 2.18: To use the } character as text.

And the following logical operators can be used to group expressions:

Operator	Description
<code>and</code>	Deprecated: use <code>&&</code> instead. The logical and operator is used to group two expressions.
<code>or</code>	Deprecated: use <code> </code> instead. The logical or operator is used to group two expressions.
<code>&&</code>	Camel 2.9: The logical and operator is used to group two expressions.
<code> </code>	Camel 2.9: The logical or operator is used to group two expressions.

Using and,or operators

In **Camel 2.4 and older** the and or or can only be used **once** in a simple language expression.

From **Camel 2.5:** you can use these operators multiple times.

The syntax for AND is:

```
java${leftValue} OP rightValue and ${leftValue} OP rightValue
```

And the syntax for OR is:

```
java${leftValue} OP rightValue or ${leftValue} OP rightValue
```

Some examples:

```
java// exact equals match simple("${in.header.foo} == 'foo') // ignore case when comparing, so if the header has value FOO this will match simple("${in.header.foo} =~ 'foo') // here Camel will type convert '100' into the type of in.header.bar and if it is an Integer '100' will also be converter to an Integer simple("${in.header.bar} == '100") simple("${in.header.bar} == 100") // 100 will be converter to the type of in.header.bar so we can do > comparison simple("${in.header.bar} > 100") Comparing with different types When you compare with different types such as String and int, then you have to take a bit care. Camel will use the type from the left hand side as first priority. And fallback to the right hand side type if both values couldn't be compared based on that type. This means you can flip the values to enforce a specific type. Suppose the bar value above is a String. Then you can flip the equation:  
javasimple("100 < ${in.header.bar}")
```

which then ensures the int type is used as first priority.

This may change in the future if the Camel team improves the binary comparison operations to prefer numeric types over String based. It's most often the String type which causes problem when comparing with numbers.

```
java// testing for null simple("${in.header.baz} == null") // testing for not null simple("${in.header.baz} != null")
```

And a bit more advanced example where the right value is another expression,

```
javasimple("${in.header.date} == ${date:now:yyyyMMdd}") simple("${in.header.type} == ${bean:orderService? method=getOrderType}")
```

And an example with contains, testing if the title contains the word Camel:

```
javasimple("${in.header.title} contains 'Camel")
```

And an example with regex, testing if the number header is a four digit value:

```
javasimple("${in.header.number} regex '\\d{4}'")
```

And finally an example if the header equals any of the values in the list. Each element must be separated by comma, and no space around. This also works for numbers etc, as Camel will convert each element into the type of the left hand side.

```
javasimple("${in.header.type} in 'gold,silver")
```

And for all the last three we also support the negate test using not:

```
javasimple("${in.header.type} not in 'gold,silver'")
And you can test if the type is a certain instance, e.g., for instance a String:
javasimple("${in.header.type} is 'java.lang.String'")
We have added a shorthand for all java.lang types so you can write it as:
javasimple("${in.header.type} is 'String'")
Ranges are also supported. The range interval requires numbers and both from and end are inclusive. For instance to test whether a value is between 100 and 199:
javasimple("${in.header.number} range 100..199")
Notice we use .. in the range without spaces. It is based on the same syntax as Groovy.
```

From **Camel 2.9**: the range value must be in single quotes:
 javasimple("\${in.header.number} range '100..199'") Can be used in Spring XML.
 As the Spring XML does not have all the power as the Java DSL with all its various builder methods, you have to resort to use some other languages for testing with simple operators. Now you can do this with the simple language. In the sample below we want to test if the header is a widget order:
`xml<from uri="seda:orders"> <filter> <simple>${in.header.type} == 'widget'</simple> <to uri="bean:orderService?method=handleWidget"/> </filter> </from>`

Using and / or

If you have two expressions you can combine them with the and or or operator.

Camel 2.9 onwards

Use && or ||

For instance:

```
javasimple("${in.header.title} contains 'Camel' and ${in.header.type} == 'gold'")
```

And of course the or is also supported. The sample would be:

```
javasimple("${in.header.title} contains 'Camel' or ${in.header.type} == 'gold'")
```

Note: currently and or or can only be used **once** in a simple language expression. This might change in the future. So you cannot do:

```
javasimple("${in.header.title} contains 'Camel' and ${in.header.type} == 'gold' and ${in.header.number} range 100..200")
```

Samples

In the Spring XML sample below we filter based on a header value:

```
xml<from uri="seda:orders"> <filter> <simple>${in.header.foo}</simple> <to uri="mock:fooOrders"/> </filter> </from>
```

The Simple language can be used for the predicate test above in the Message Filter pattern, where we test if the in message has a foo header (a header with the key foo exists). If the expression evaluates to true then the message is routed to the mock : fooreders endpoint, otherwise it is lost in the deep blue sea 😊.

The same example in Java DSL:

```
javafrom("seda:orders") .filter().simple("${in.header.foo}") .to("seda:fooreders");
```

You can also use the simple language for simple text concatenations such as:

```
javafrom("direct:hello") .transform().simple("Hello ${in.header.user} how are you?") .to("mock:reply");
```

Notice that we must use \${} placeholders in the expression now to allow Camel to parse it correctly.

And this sample uses the date command to output current date.

```
javafrom("direct:hello") .transform().simple("The today is ${date:now:yyyyMMdd} and it is a great day.") .to("mock:reply");
```

And in the sample below we invoke the bean language to invoke a method on a bean to be included in the returned string:

```
javafrom("direct:order") .transform().simple("Orderid: ${bean:orderIdGenerator}") .to("mock:reply");
```

Where orderIdGenerator is the id of the bean registered in the Registry. If using Spring then it is the Spring bean id.

If we want to declare which method to invoke on the order id generator bean we must prepend .method name such as below where we invoke the generateId method.

```
javafrom("direct:order") .transform().simple("Orderid: ${bean:orderIdGenerator.generateId}") .to("mock:reply");
```

We can use the ?method=methodname option that we are familiar with the Bean component itself:

```
javafrom("direct:order") .transform().simple("Orderid: ${bean:orderIdGenerator?method=generateId}") .to("mock:reply");
```

From **Camel 2.3**: you can also convert the body to a given type, for example to ensure that it is a String you can do:

```
xml<transform> <simple>Hello ${bodyAs(String)} how are you?</simple> </transform>
```

There are a few types which have a shorthand notation, so we can use String instead of java.lang.String. These are:

```
byte[], String, Integer, Long. All other types must use their FQN name, e.g. org.w3c.dom.Document.
```

It is also possible to lookup a value from a header Map in **Camel 2.3**:

```
xml<transform> <simple>The gold value is ${header.type[gold]}</simple> </transform>
```

In the code above we lookup the header with name type and regard it as a java.util.Map and we then lookup with the key gold and return the value. If the header is not convertible to Map an exception is thrown. If the header with name type does not exist null is returned.

From **Camel 2.9**: you can nest functions, such as shown below:

```
xml<setHeader headerName="myHeader"> <simple>${properties:${header.someKey}}</simple> </setHeader>
```

Referring to Constants or Enums

Available from Camel 2.11

Suppose you have an enum for customers:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/Customer.java}And in a Content Based Router we can use the Simple language to refer to this enum, to check the message which enum it matches.{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/CBRSimpleTypeTest.java}

Using New Lines or Tabs in XML DSLs

Available from Camel 2.9.3

From Camel 2.9.3: it is easier to specify new lines or tabs in XML DSLs as you can escape the value now
xml<transform> <simple>The following text\nis on a new line</simple> </transform>

Leading and Trailing Whitespace Handling

Available from Camel 2.10.0

From Camel 2.10.0: the trim attribute of the expression can be used to control whether the leading and trailing whitespace characters are removed or preserved. The default of trim=true removes all whitespace characters.

xml<setBody> <simple trim="false">You get some trailing whitespace characters. </simple> </setBody>

Setting the Result Type

Available from Camel 2.8

You can now provide a result type to the Simple expression, which means the result of the evaluation will be converted to the desired type. This is most usable to define types such as boolean's, integer's, etc.

For example to set a header as a boolean type you can do:

.setHeader("cool", simple("true", Boolean.class))

And in XML DSL

xml<setHeader headerName="cool"> <!-- use resultType to indicate that the type should be a java.lang.Boolean --> <simple resultType="java.lang.Boolean">true</simple> </setHeader>

Changing Function Start and End Tokens

Available from Camel 2.9.1

You can configure the function start and end tokens - \${} using the setters changeFunctionStartToken and changeFunctionEndToken on SimpleLanguage, using Java code. From Spring XML you can define a <bean> tag with the new changed tokens in the properties as shown below:

xml<!-- config Simple to use custom prefix/suffix tokens --> <bean id="simple" class="org.apache.camel.language.simple.SimpleLanguage"> <property name="functionStartToken" value="["/> <property name="functionEndToken" value="]" /> </bean>

In the example above we use [] as the changed tokens. Notice by changing the start/end token you change those in all the Camel applications which share the same camel-core on their classpath. For example in an OSGi server this may affect many applications, where as a Web Application as a WAR file it only affects the Web Application.

Loading Script from External Resource

Available from Camel 2.11

You can externalize the script and have Camel load it from a resource such as: classpath:, file:, or http:. This is done using the following syntax: resource:scheme:location, e.g., to refer to a file on the classpath you can do:
java.setHeader("myHeader").simple("resource:classpath:mysimple.txt")

Setting Spring beans to Exchange properties

Available from Camel 2.6

You can set a spring bean into an exchange property as shown below:

xml<bean id="myBeanId" class="my.package.MyCustomClass"/> <route> <!-- ... --> <setProperty propertyName="monitoring.message"> <simple>ref:myBeanId</simple> </setProperty> <!-- ... --> </route>

Dependencies

The Simple language is part of camel-core.

FILE EXPRESSION LANGUAGE

File language is now merged with Simple language

From Camel 2.2: the file language is now merged with Simple language which means you can use all the file syntax directly within the simple language.

The File Expression Language is an extension to the Simple language, adding file related capabilities. These capabilities are related to common use cases working with file path and names. The goal is to allow expressions to be used with the File and FTP components for setting dynamic file patterns for both consumer and producer.

Syntax

This language is an **extension** to the Simple language so the Simple syntax applies also. So the table below only lists the additional. By contrast to the Simple language, the File Language also supports the use of Constant expressions to enter a fixed filename, for example.

All the file tokens use the same expression name as the method on the `java.io.File` object. For example: `file:absolute` refers to the `java.io.File.getAbsolutePath()` method.

Note: not all expressions are supported by the current Exchange. For example, the FTP component supports some of the options, whereas the File component supports all of them.

Expression	Type	File Consumer	File Producer	FTP Consumer	FTP Producer	Description
<code>date:command:pattern</code>	String	yes	yes	yes	yes	For date formatting using the <code>java.text.SimpleDateFormat</code> patterns which is an extension to the Simple language. Additional command is: <code>file</code> (consumers only) for the last modified timestamp of the file. Note: all the commands from the Simple language can also be used.
<code>file:absolute</code>	Boolean	yes	no	no	no	Refers to whether the file is regarded as absolute or relative.
<code>file:absolute.path</code>	String	yes	no	no	no	Refers to the absolute file path.
<code>file:exists</code>	Boolean	yes	no	no	no	
<code>file:length</code>	Long	yes	no	yes	no	Refers to the file length returned as a Long type.
<code>file:modified</code>	Date	yes	no	yes	no	Refers to the file last modified returned as a date type.
<code>filename</code>	String	yes	no	yes	no	Refers to the file name (is relative to the starting directory, see note below).
<code>filename.ext</code>	String	yes	no	yes	no	Camel 2.3: refers to the file extension only.
<code>filename.ext:single</code>	String	yes	no	yes	no	Camel 2.14.2/2.15.3: refers to the file extension. If the file extension has multiple dots, then this expression strips and only returns the last part.
<code>filename.noext</code>	String	yes	no	yes	no	Refers to the file name with no extension (is relative to the starting directory, see note below).
<code>filename.noext:single</code>	String	yes	no	yes	no	Camel 2.14.2/2.15.3: refers to the file name with no extension (is relative to the starting directory, see note below). If the file extension has multiple dots, then this expression strips only the last part, and keep the others.
<code>fileonlyname</code>	String	yes	no	yes	no	Refers to the file name only with no leading paths.
<code>fileonlyname.noext</code>	String	yes	no	yes	no	Refers to the file name only with no extension and with no leading paths.
<code>fileparent</code>	String	yes	no	yes	no	Refers to the file parent.
<code>filepath</code>	String	yes	no	yes	no	Refers to the file path.
<code>filesize</code>	Long	yes	no	yes	no	Camel 2.5: refers to the file length returned as a Long type.

File Token Example

Relative Paths

We have a `java.io.File` handle for the file `hello.txt` in the following *relative* directory: `.\filelanguage\test`. And we configure our endpoint to use this starting directory `.\filelanguage`.

The file tokens returned are:

Expression	Returns
<code>file:absolute</code>	false
<code>file:absolute.path</code>	<code>\workspace\camel\camel-core\target\filelanguage\test\hello.txt</code>
<code>file:ext</code>	<code>txt</code>
<code>file:name</code>	<code>test\hello.txt</code>
<code>file:name.ext</code>	<code>txt</code>
<code>file:name.noext</code>	<code>test\hello</code>
<code>file:onlyname</code>	<code>hello.txt</code>
<code>file:onlyname.noext</code>	<code>hello</code>
<code>file:parent</code>	<code>filelanguage\test</code>
<code>file:path</code>	<code>filelanguage\test\hello.txt</code>

Absolute Paths

We have a `java.io.File` handle for the file `hello.txt` in the following *absolute* directory: `\workspace\camel\camel-core\target\filelanguage\test`. And we configure our endpoint to use the absolute starting directory: `\workspace\camel\camel-core\target\filelanguage`.

The file tokens return are:

Expression	Returns
<code>file:absolute</code>	true
<code>file:absolute.path</code>	<code>\workspace\camel\camel-core\target\filelanguage\test\hello.txt</code>
<code>file:ext</code>	<code>txt</code>
<code>file:name</code>	<code>test\hello.txt</code>
<code>file:name.ext</code>	<code>txt</code>

file:name.noext	test\hello
file:onlyname	hello.txt
file:onlyname.noext	hello
file:parent	\workspace\camel\camel-core\target\filelanguage\test
file:path	\workspace\camel\camel-core\target\filelanguage\test\hello.txt

Examples

You can enter a fixed Constant expression such as myfile.txt:

```
fileName="myfile.txt"
```

Lets assume we use the file consumer to read files and want to move the read files to backup folder with the current date as a sub folder. This can be achieved using an expression like:

```
fileName="backup/${date:now:yyyyMMdd}/${file:name.noext}.bak"
```

relative folder names are also supported so suppose the backup folder should be a sibling folder then you can append .. as:

```
fileName="../backup/${date:now:yyyyMMdd}/${file:name.noext}.bak"
```

As this is an extension to the Simple language we have access to all the goodies from this language also, so in this use case we want to use the `in.header.type` as a parameter in the dynamic expression:

```
fileName="../backup/${date:now:yyyyMMdd}/type-${in.header.type}/backup-of-${file:name.noext}.bak"
```

If you have a custom Date you want to use in the expression then Camel supports retrieving dates from the message header.

```
fileName="orders/order-${in.header.customerId}-${date:in.header.orderDate:yyyyMMdd}.xml"
```

And finally we can also use a bean expression to invoke a POJO class that generates some String output (or convertible to String) to be used:

```
fileName="uniquefile-${bean:myguidgenerator.generateid}.txt"
```

And of course all this can be combined in one expression where you can use the File Language, Simple and the Bean language in one combined expression. This is pretty powerful for those common file path patterns.

Using Spring's PropertyPlaceholderConfigurer with the File Component

In Camel you can use the File Language directly from the Simple language which makes a Content Based Router easier to do in Spring XML, where we can route based on file extensions as shown below:

```
<from uri="file://input/orders"/>
<choice>
  <when>
    <simple>${file:ext} == &#39;txt'&#39;</simple>
    <to uri="bean:orderService?method=handleTextFiles"/>
  </when>
  <when>
    <simple>${file:ext} == &#39;xml'&#39;</simple>
    <to uri="bean:orderService?method=handleXmlFiles"/>
  </when>
  <otherwise>
    <to uri="bean:orderService?method=handleOtherFiles"/>
  </otherwise>
</choice>
```

If you use the `fileName` option on the File endpoint to set a dynamic filename using the File Language then make sure you use the alternative syntax (available from Camel 2.5) to avoid clashing with Spring's `PropertyPlaceholderConfigurer`.

```
Listing 1. bundle-context.xml

<bean id="propertyPlaceholder" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location" value="classpath:bundle-context.cfg"/>
</bean>

<bean id="sampleRoute" class="SampleRoute">
  <property name="fromEndpoint" value="${fromEndpoint}"/>
  <property name="toEndpoint" value="${toEndpoint}"/>
</bean>
```

```
Listing 1. bundle-context.cfg

fromEndpoint=activemq:queue:test
toEndpoint=file:///Fileroute/out?fileName=test-$simple{date:now:yyyyMMdd}.txt
```

Notice how we use the `$simple{}` syntax in the `toEndpoint` above. If you don't do this, they will clash and Spring will throw an exception:

```
org.springframework.beans.factory.BeanDefinitionStoreException:  
Invalid bean definition with name &lt;sampleRoute&gt; defined in class path resource [bundle-context.xml]:  
Could not resolve placeholder &lt;date:now:yyyyMdd&gt;
```

Dependencies

The File language is part of `camel-core`.

SQL LANGUAGE

The SQL support is added by JoSQL and is primarily used for performing SQL queries on in-memory objects. If you prefer to perform actual database queries then check out the JPA component.

Looking for the SQL component

Camel has both a SQL language and a SQL Component. This page is about the SQL language. Click on SQL Component if you are looking for the component instead.

To use SQL in your camel routes you need to add the dependency on `camel-josql` which implements the SQL language.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

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

Camel supports SQL to allow an Expression or Predicate to be used in the DSL or Xml Configuration. For example you could use SQL to create an Predicate in a Message Filter or as an Expression for a Recipient List.

```
from("queue:foo").setBody().sql("select * from MyType").to("queue:bar")
```

And the spring DSL:

```
<from uri="queue:foo"/>  
<setBody>  
<sql>select * from MyType</sql>  
</setBody>  
<to uri="queue:bar"/>
```

Variables

Variable	Type	Description
exchange	Exchange	the Exchange object
in	Message	the exchange.in message
out	Message	the exchange.out message
the property key	Object	the Exchange properties
the header key	Object	the exchange.in headers
the variable key	Object	if any additional variables is added using setvariables method

Loading script from external resource

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as "classpath:", "file:", or "http:". This is done using the following syntax: "resource:scheme:location", eg to refer to a file on the classpath you can do:

```
.setHeader("myHeader").sql("resource:classpath:mysql.sql")
```

XPATH

Camel supports XPath to allow an Expression or Predicate to be used in the DSL or Xml Configuration. For example you could use XPath to create an Predicate in a Message Filter or as an Expression for a Recipient List.

Streams

If the message body is stream based, which means the input is received by Camel as a stream, then you will only be able to read the content of the stream **once**. Oftentimes when using XPath as Message Filter or Content Based Router the data will be accessed multiple times. Therefore use Stream caching or convert the message body to a `String` beforehand. This makes it safe to be re-read multiple times.

```
from("queue:foo").filter().xpath("//foo")).to("queue:bar") from("queue:foo") .choice().xpath("//foo").to("queue:bar")
.otherwise().to("queue:others");
```

Namespaces

You can easily use namespaces with XPath expressions using the Namespaces helper class.

```
{snippet:id=example|lang=java|url=camel/trunk/camel-
core/src/test/java/org/apache/camel/processor/XPathWithNamespacesFilterTest.java}
```

Variables

Variables in XPath is defined in different namespaces. The default namespace is `http://camel.apache.org/schema/spring`.

Namespace URI	Local part	Type	Description
<code>http://camel.apache.org/xml/in/</code>	<code>in</code>	Message	The exchange.in message.
<code>http://camel.apache.org/xml/out/</code>	<code>out</code>	Message	The exchange.out message.
<code>http://camel.apache.org/xml/function/</code>	<code>functions</code>	Object	Camel 2.5: Additional functions.
<code>http://camel.apache.org/xml/variables/environment-variables</code>	<code>env</code>	Object	OS environment variables.
<code>http://camel.apache.org/xml/variables/system-properties</code>	<code>system</code>	Object	Java System properties.
<code>http://camel.apache.org/xml/variables/exchange-property</code>		Object	The exchange property.

Camel will resolve variables according to either:

- namespace given
- no namespace given

Namespace Given

If the namespace is given then Camel is instructed exactly what to return. However when resolving either IN or OUT Camel will try to resolve a header with the given local part first, and return it. If the local part has the value body then the body is returned instead.

No Namespace Given

If there is no namespace given then Camel resolves only based on the local part. Camel will try to resolve a variable in the following steps:

- From variables that has been set using the `variable(name, value)` fluent builder.
- From `message.in.header` if there is a header with the given key.
- From `exchange.properties` if there is a property with the given key.

Functions

Camel adds the following XPath functions that can be used to access the exchange:

Function	Argument	Type	Description
<code>in:body</code>	none	Object	Will return the IN message body.
<code>in:header</code>	the header name	Object	Will return the IN message header.
<code>out:body</code>	none	Object	Will return the OUT message body.
<code>out:header</code>	the header name	Object	Will return the OUT message header.
<code>function:properties</code>	key for property	String	Camel 2.5: To lookup a property using the Properties component (property placeholders).
<code>function:simple</code>	simple expression	Object	Camel 2.5: To evaluate a Simple expression.

Note: `function:properties` and `function:simple` is not supported when the return type is a `Nodeset`, such as when using with a Splitter EIP.

Here's an example showing some of these functions in use.

```
{snippet:id=ex|lang=java|url=camel/trunk/camel-
core/src/test/java/org/apache/camel/language/XPathFunctionTest.java}And the new functions introduced in Camel 2.5:
```

```
{snippet:id=ex|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/builder/xml/XPathFunctionsTest.java}
```

Using XML Configuration

If you prefer to configure your routes in your Spring XML file then you can use XPath expressions as follows

```
xml<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=" http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
beans.xsd http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd"> <camelContext
id="camel" xmlns="http://activemq.apache.org/camel/schema/spring"> xmlns:foo="http://example.com/person"> <route> <from
uri="activemq:MyQueue"/> <filter> <xpath>/foo:person[@name='James']</xpath> <to uri="mqseries:SomeOtherQueue"/> </filter>
</route> </camelContext> </beans>
```

Notice how we can reuse the namespace prefixes, foo in this case, in the XPath expression for easier namespace based XPath expressions! See also this discussion on the mailinglist about using your own namespaces with XPath.

Setting the Result Type

The XPath expression will return a result type using native XML objects such as `org.w3c.dom.NodeList`. But many times you want a result type to be a `String`. To do this you have to instruct the XPath which result type to use.

In Java DSL:

```
jaxxpath("//foo:person/@id", String.class)
In Spring DSL you use the resultType attribute to provide a fully qualified classname:
xml<xpath resultType="java.lang.String">/foo:person/@id</xpath>
In @XPath:
```

Available as of Camel 2.1

```
java@XPath(value = "concat('foo-',//order/name/)", resultType = String.class) String name)
```

Where we use the XPath function concat to prefix the order name with foo-. In this case we have to specify that we want a `String` as result type so the concat function works.

Using XPath on Headers

Available as of Camel 2.11

Some users may have XML stored in a header. To apply an XPath statement to a header's value you can do this by defining the headerName attribute.

```
In XML DSL:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-test-
blueprint/src/test/resources/org/apache/camel/test/blueprint/xpath/XPathHeaderNameTest.xml}And in Java DSL you specify
the headerName as the 2nd parameter as shown:
jaxxpath("//invoice/@orderType = 'premium'", "invoiceDetails")
```

Examples

Here is a simple example using an XPath expression as a predicate in a Message Filter:{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/XPathFilterTest.java}If you have a standard set of namespaces you wish to work with and wish to share them across many different XPath expressions you can use the NamespaceBuilder as shown in this example:{snippet:id=example|lang=java|url=camel/trunk/camel-
core/src/test/java/org/apache/camel/processor/XPathWithNamespaceBuilderFilterTest.java}In this sample we have a choice construct. The first choice evaluates if the message has a header key type that has the value camel. The 2nd choice evaluates if the message body has a name tag <name> which values is kong. If neither is true the message is routed in the otherwise block:{snippet:id=e1|lang=java|url=camel/trunk/camel-
core/src/test/java/org/apache/camel/builder/xml/XPathHeaderTest.java}And the spring XML equivalent of the route:
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-
spring/src/test/resources/org/apache/camel/spring/processor/SpringXPathHeaderTest-context.xml}

XPATH INJECTION

You can use Bean Integration to invoke a method on a bean and use various languages such as XPath to extract a value from the message and bind it to a method parameter.

The default XPath annotation has SOAP and XML namespaces available. If you want to use your own namespace URIs in an XPath expression you can use your own copy of the XPath annotation to create whatever namespace prefixes you want to use. {snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/xslt/MyXPath.java}e.g., cut and paste upper code to your own project in a different package and/or annotation name then add whatever namespace prefix/URIs you want in scope when you use your annotation on a method parameter. Then when you use your annotation on a method parameter all the namespaces you want will be available for use in your XPath expression.

Example:

```
javapublic class Foo { @MessageDriven(uri = "activemq:my.queue") public void
doSomething(@MyXPath("ns1:foo/ns2:bar/text()") String correlationID, @Body String body) { // process the inbound message
here } }
```

Using XPathBuilder Without an Exchange

Available as of Camel 2.3

You can now use the `org.apache.camel.builder.XPathBuilder` without the need for an Exchange. This comes handy if you want to use it as a helper to do custom XPath evaluations. It requires that you pass in a CamelContext since a lot of the moving parts inside the `xPathBuilder` requires access to the Camel Type Converter and hence why CamelContext is needed.

For example you can do something like this:

```
javaBoolean matches = XPathBuilder.xpath("//foo/bar/@xyz").matches(context, "<foo><bar xyz='cheese'/'></foo>");  
This will match the given predicate.
```

You can also evaluate for example as shown in the following three examples:

```
javaString name = XPathBuilder.xpath("foo/bar").evaluate(context, "<foo><bar>cheese</bar></foo>", String.class); Integer  
number = XPathBuilder.xpath("foo/bar").evaluate(context, "<foo><bar>123</bar></foo>", Integer.class); Boolean bool =  
XPathBuilder.xpath("foo/bar").evaluate(context, "<foo><bar>true</bar></foo>", Boolean.class);  
Evaluating with a String result is a common requirement and thus you can do it a bit simpler:  
String name = XPathBuilder.xpath("foo/bar").evaluate(context, "<foo><bar>cheese</bar></foo>");
```

Using Saxon with XPathBuilder

Available as of Camel 2.3

You need to add `camel-saxon` as dependency to your project. It's now easier to use Saxon with the `xPathBuilder` which can be done in several ways as shown below. Where as the latter ones are the easiest ones.

```
Using a factory:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-  
saxon/src/test/java/org/apache/camel/builder/saxon/XPathTest.java}Using the object model  
{snippet:id=e2|lang=java|url=camel/trunk/components/camel-  
saxon/src/test/java/org/apache/camel/builder/saxon/XPathTest.java}The easy  
one{snippet:id=e3|lang=java|url=camel/trunk/components/camel-  
saxon/src/test/java/org/apache/camel/builder/saxon/XPathTest.java}
```

Setting a Custom XPathFactory Using System Property

Available as of Camel 2.3

Camel now supports reading the JVM system property `javax.xml.xpath.XPathFactory` that can be used to set a custom `xPathFactory` to use.

This unit test shows how this can be done to use Saxon instead:{snippet:id=e4|lang=java|url=camel/trunk/components/camel-
saxon/src/test/java/org/apache/camel/builder/saxon/XPathTest.java}Camel will log at INFO level if it uses a non
default `xPathFactory` such as:
XPathBuilder INFO Using system property `javax.xml.xpath.XPathFactory:http://saxon.sf.net/jaxp/xpath/om` with value:
`net.sf.saxon.xpath.XPathFactoryImpl` when creating `XPathFactory`
To use Apache Xerces you can configure the system property:
`-Djavax.xml.xpath.XPathFactory=org.apache.xpath.jaxp.XPathFactoryImpl`

Enabling Saxon from Spring DSL

Available as of Camel 2.10

Similarly to Java DSL, to enable Saxon from Spring DSL you have three options:

```
Specifying the factory  
xml<xpath factoryRef="saxonFactory" resultType="java.lang.String">current-dateTime()</xpath>  
Specifying the object model  
xml<xpath objectModel="http://saxon.sf.net/jaxp>xpath/om" resultType="java.lang.String">current-dateTime()</xpath>  
Shortcut  
xml<xpath saxon="true" resultType="java.lang.String">current-dateTime()</xpath>
```

Namespace Auditing to Aid Debugging

Available as of Camel 2.10

A large number of XPath-related issues that users frequently face are linked to the usage of namespaces. You may have some misalignment between the namespaces present in your message and those that your XPath expression is aware of or referencing. XPath predicates or expressions that are unable to locate the XML elements and attributes due to namespaces issues may simply look like "they are not working", when in reality all there is to it is a lack of namespace definition.

Namespaces in XML are completely necessary, and while we would love to simplify their usage by implementing some magic or voodoo to wire namespaces automatically, truth is that any action down this path would disagree with the standards and would greatly hinder interoperability.

Therefore, the utmost we can do is assist you in debugging such issues by adding two new features to the XPath Expression Language and are thus accessible from both predicates and expressions.

Logging the Namespace Context of Your XPath Expression/Predicate

Every time a new XPath expression is created in the internal pool, Camel will log the namespace context of the expression under the `org.apache.camel.builder.xml.XPathBuilder` logger. Since Camel represents Namespace Contexts in a hierarchical fashion (parent-child relationships), the entire tree is output in a recursive manner with the following format:

```
[me: {prefix -> namespace}, {prefix -> namespace}], [parent: [me: {prefix -> namespace}, {prefix -> namespace}], [parent: [me: {prefix -> namespace}]]]
```

Any of these options can be used to activate this logging:

1. Enable TRACE logging on the `org.apache.camel.builder.xml.XPathBuilder` logger, or some parent logger such as `org.apache.camel` or the root logger.
2. Enable the `logNamespaces` option as indicated in Auditing Namespaces, in which case the logging will occur on the INFO level.

AuditingNamespaces

Auditing namespaces

Camel is able to discover and dump all namespaces present on every incoming message before evaluating an XPath expression, providing all the richness of information you need to help you analyse and pinpoint possible namespace issues. To achieve this, it in turn internally uses another specially tailored XPath expression to extract all namespace mappings that appear in the message, displaying the prefix and the full namespace URI(s) for each individual mapping.

Some points to take into account:

- The implicit XML namespace (`xmlns:xml="http://www.w3.org/XML/1998/namespace"`) is suppressed from the output because it adds no value.
- Default namespaces are listed under the `DEFAULT` keyword in the output.
- Keep in mind that namespaces can be remapped under different scopes. Think of a top-level 'a' prefix which in inner elements can be assigned a different namespace, or the default namespace changing in inner scopes. For each discovered prefix, all associated URIs are listed.

You can enable this option in Java DSL and Spring DSL.

Java DSL:

```
javaXPathBuilder.xpath("/foo:person/@id", String.class).logNamespaces()
```

Spring DSL:

```
xml<xpath logNamespaces="true" resultType="String">/foo:person/@id</xpath>
```

The result of the auditing will be appear at the INFO level under the `org.apache.camel.builder.xml.XPathBuilder` logger and will look like the following:

```
2012-01-16 13:23:45,878 [stSaxonWithFlag] INFO XPathBuilder - Namespaces discovered in message: {xmlns:a=[http://apache.org/camel], DEFAULT=[http://apache.org/default], xmlns:b=[http://apache.org/camelA, http://apache.org/camelB]}
```

Loading Script from External Resource

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as: `classpath:`, `file:` or `http:`. This is done using the following syntax: `resource:scheme:location`, e.g., to refer to a file on the classpath you can do:
`.setHeader("myHeader").xpath("resource:classpath:myxpath.txt", String.class)`

Dependencies

The XPath language is part of camel-core.

XQUERY

Camel supports XQuery to allow an Expression or Predicate to be used in the DSL or Xml Configuration. For example you could use XQuery to create an Predicate in a Message Filter or as an Expression for a Recipient List.

Options

confluenceTableSmall

Name	Default Value	Description
<code>allowStAX</code>	false	<code>Camel 2.8.3/2.9:</code> Whether to allow using StAX as the <code>javax.xml.transform.Source</code> .

Examples

```
from("queue:foo").filter().xquery("//foo").to("queue:bar")
```

You can also use functions inside your query, in which case you need an explicit type conversion (or you will get a `org.w3c.dom.DOMException: HIERARCHY_REQUEST_ERR`) by passing the Class as a second argument to the `xquery()`

```
method.
from("direct:start").recipientList().xquery("concat('mock:foo.', /person/@city)", String.class);
```

Variables

The IN message body will be set as the `contextItem`. Besides this these Variables is also added as parameters:

Variable	Type	Description
exchange	Exchange	The current Exchange
in.body	Object	The In message's body
out.body	Object	The OUT message's body (if any)
in.headers.*	Object	You can access the value of exchange.in.headers with key foo by using the variable which name is in.headers.foo
out.headers.*	Object	You can access the value of exchange.out.headers with key foo by using the variable which name is out.headers.foo variable
key name	Object	Any exchange.properties and exchange.in.headers and any additional parameters set using <code>setParameters(Map)</code> . These parameters is added with they own key name, for instance if there is an IN header with the key name foo then its added as foo .

Using XML configuration

If you prefer to configure your routes in your Spring XML file then you can use XPath expressions as follows

```
xml<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:foo="http://example.com/person" xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd http://camel.apache.org/schema/spring
  http://camel.apache.org/schema/spring/camel-spring.xsd"> <camelContext id="camel"
  xmlns="http://activemq.apache.org/camel/schema/spring"> <route> <from uri="activemq:MyQueue"/> <filter>
  <xquery>/foo:person[@name='James'][/xquery> <to uri="mqseries:SomeOtherQueue"/> </filter> </route> </camelContext>
</beans>
```

Notice how we can reuse the namespace prefixes, **foo** in this case, in the XPath expression for easier namespace based XQuery expressions!

When you use functions in your XQuery expression you need an explicit type conversion which is done in the xml configuration via the `@type` attribute:

```
xml <xquery type="java.lang.String">concat('mock:foo.', /person/@city)</xquery>
```

Using XQuery as transformation

We can do a message translation using transform or setBody in the route, as shown below:

```
from("direct:start").transform().xquery("/people/person");
```

Notice that xquery will use DOMResult by default, so if we want to grab the value of the person node, using `text()` we need to tell xquery to use String as result type, as shown:

```
from("direct:start").transform().xquery("/people/person/text()", String.class);
```

Using XQuery as an endpoint

Sometimes an XQuery expression can be quite large; it can essentially be used for Templating. So you may want to use an XQuery Endpoint so you can route using XQuery templates.

The following example shows how to take a message of an ActiveMQ queue (MyQueue) and transform it using XQuery and send it to MQSeries.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring"> <route> <from uri="activemq:MyQueue"/> <to
uri="xquery:com/acme/someTransform.xquery"/> <to uri="mqseries:SomeOtherQueue"/> </route> </camelContext>
```

Examples

Here is a simple example using an XQuery expression as a predicate in a Message Filter

```
{snippet:id=example||lang=java|url=camel/trunk/components/camel-
saxon/src/test/java/org/apache/camel/builder/saxon/XQueryFilterTest.java}
```

This example uses XQuery with namespaces as a predicate in a Message Filter

```
{snippet:id=example||lang=java|url=camel/trunk/components/camel-
saxon/src/test/java/org/apache/camel/builder/saxon/XQueryWithNamespacesFilterTest.java}
```

Learning XQuery

XQuery is a very powerful language for querying, searching, sorting and returning XML. For help learning XQuery try these tutorials

- Mike Kay's XQuery Primer
- the W3Schools XQuery Tutorial

You might also find the XQuery function reference useful

Loading script from external resource

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as "classpath:", "file:", or "http:". This is done using the following syntax: "resource:scheme:location", eg to refer to a file on the classpath you can do: .setHeader("myHeader").xquery("resource:classpath:myxpath.txt", String.class)

Dependencies

To use XQuery in your camel routes you need to add the a dependency on **camel-saxon** which implements the XQuery language.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-saxon</artifactId> <version>x.x.x</version>
</dependency>
```

Data Format Appendix

DATA FORMAT

Camel supports a pluggable DataFormat to allow messages to be marshalled to and from binary or text formats to support a kind of Message Translator.

The following data formats are currently supported:

- Standard JVM object marshalling
 - Serialization
 - String
- Object marshalling
 - Avro
 - Boon
 - Hessian
 - JSON
 - Protobuf
 - YAML
- Object/XML marshalling
 - Castor
 - JAXB
 - XmlBeans
 - XStream
 - JIBX
 - Jackson XML
- Object/XML/Webservice marshalling
 - SOAP
- Direct JSON / XML marshalling
 - XmlJson
- Flat data structure marshalling
 - BeanIO
 - Bindy
 - CSV
 - EDI
 - Flatpack DataFormat
 - uniVocity-parsers formats
- Domain specific marshalling
 - HL7 DataFormat
- Compression
 - GZip data format
 - Zip DataFormat
 - Zip File DataFormat
 - LZF Data Format
 - Tar DataFormat
- Security
 - Crypto
 - PGP
 - XMLSecurity DataFormat
- Misc.
 - Base64
 - Custom DataFormat - to use your own custom implementation
 - MIME-Multipart
 - RSS

- TidyMarkup
- Syslog
- iCal
- Barcode - to read and generate barcodes (QR-Code, PDF417, ...)

And related is the following:

- DataFormat Component for working with Data Formats as if it was a regular Component supporting Endpoints and URIs.
- Dozer Type Conversion using Dozer for type converting POJOs

Unmarshalling

If you receive a message from one of the Camel Components such as File, HTTP or JMS you often want to unmarshal the payload into some bean so that you can process it using some Bean Integration or perform Predicate evaluation and so forth. To do this use the **unmarshal** word in the DSL in Java or the XML Configuration.

For example

```
DataFormat jaxb = new JaxbDataFormat("com.acme.model");
from("activemq:My.Queue")
    unmarshal(jaxb)
    to("mqseries:Another.Queue");
```

The above uses a named DataFormat of *jaxb* which is configured with a number of Java package names. You can if you prefer use a named reference to a data format which can then be defined in your Registry such as via your Spring XML file.

You can also use the DSL itself to define the data format as you use it. For example the following uses Java serialization to unmarshal a binary file then send it as an ObjectMessage to ActiveMQ

```
from("file://foo/bar").
    unmarshal().serialization().
    to("activemq:Some.Queue");
```

Marshalling

Marshalling is the opposite of unmarshalling, where a bean is marshalled into some binary or textual format for transmission over some transport via a Camel Component. Marshalling is used in the same way as unmarshalling above; in the DSL you can use a DataFormat instance, you can configure the DataFormat dynamically using the DSL or you can refer to a named instance of the format in the Registry.

The following example unmarshals via serialization then marshals using a named JAXB data format to perform a kind of Message Translator

```
from("file://foo/bar").
    unmarshal().serialization().
    marshal("jaxb").
    to("activemq:Some.Queue");
```

Using Spring XML

This example shows how to configure the data type just once and reuse it on multiple routes

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
        <jaxb id="myJAXB" prettyPrint="true" contextPath="org.apache.camel.example"/>
    </dataFormats>

    <route>
        <from uri="direct:start"/>
        <marshal ref="myJAXB"/>
        <to uri="direct:marshalled"/>
    </route>
    <route>
        <from uri="direct:marshalled"/>
        <unmarshal ref="myJAXB"/>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

You can also define reusable data formats as Spring beans

```
<bean id="myJAXB" class="org.apache.camel.model.dataformat.JaxbDataFormat">
    <property name="prettyPrint" value="true"/>
    <property name="contextPath" value="org.apache.camel.example"/>
</bean>
```

SERIALIZATION

Serialization is a Data Format which uses the standard Java Serialization mechanism to unmarshal a binary payload into Java objects or to marshal Java objects into a binary blob.

For example the following uses Java serialization to unmarshal a binary file then send it as an ObjectMessage to ActiveMQ

```
from("file:///foo/bar").
  unmarshal().serialization().
  to("activemq:Some.Queue");
```

Dependencies

This data format is provided in **camel-core** so no additional dependencies is needed.

JAXB

JAXB is a Data Format which uses the JAXB2 XML marshalling standard which is included in Java 6 to unmarshal an XML payload into Java objects or to marshal Java objects into an XML payload.

Using the Java DSL

For example the following uses a named DataFormat of `jaxb` which is configured with a number of Java package names to initialize the JAXBContext.

```
DataFormat jaxb = new JaxbDataFormat("com.acme.model"); from("activemq:My.Queue"). unmarshal(jaxb).
to("mqseries:Another.Queue");
```

You can if you prefer use a named reference to a data format which can then be defined in your Registry such as via your Spring XML file. e.g.

```
from("activemq:My.Queue"). unmarshal("myJaxbDataType"). to("mqseries:Another.Queue");
```

Using Spring XML

The following example shows how to use JAXB to unmarshal using Spring configuring the `jaxb` data type{snippet:id=example|lang=xml|url=camel/trunk/components/camel-jaxb/src/test/resources/org/apache/camel/example/springDataFormat.xml}This example shows how to configure the data type just once and reuse it on multiple routes.{snippet:id=example|lang=xml|url=camel/trunk/components/camel-jaxb/src/test/resources/org/apache/camel/example/marshalAndUnmarshalWithRef.xml}

Multiple context paths

It is possible to use this data format with more than one context path. You can specify context path using : as separator, for example `com.mycompany:com.mycompany2`. Note that this is handled by JAXB implementation and might change if you use different vendor than RI.

Partial marshalling/unmarshalling

This feature is new to Camel 2.2.0.

JAXB 2 supports marshalling and unmarshalling XML tree fragments. By default JAXB looks for `@XmlRootElement` annotation on given class to operate on whole XML tree. This is useful but not always - sometimes generated code does not have `@XmlRootElement` annotation, sometimes you need unmarshal only part of tree.

In that case you can use partial unmarshalling. To enable this behaviour you need set property `partClass`. Camel will pass this class to JAXB's unmarshaller.{snippet:id=example|lang=xml|url=camel/trunk/components/camel-jaxb/src/test/resources/org/apache/camel/example/springDataFormatPartial.xml}For marshalling you have to add `partNamespace` attribute with QName of destination namespace. Example of Spring DSL you can find above.

Fragment

This feature is new to Camel 2.8.0.

`JaxbDataFormat` has new property `fragment` which can set the the `Marshaller.JAXB_FRAGMENT` encoding property on the JAXB Marshaller. If you don't want the JAXB Marshaller to generate the XML declaration, you can set this option to be true. The default value of this property is false.

Ignoring the NonXML Character

This feature is new to Camel 2.2.0.

`JaxbDataFormat` supports to ignore the NonXML Character, you just need to set the `filterNonXmlChars` property to be true, `JaxbDataFormat` will replace the NonXML character with " " when it is marshaling or unmarshaling the message. You can also do it by setting the Exchange property `Exchange.FILTER_NON_XML_CHARS`.

	JKD 1.5	JKD 1.6+
Filtering in use	StAX API and implementation	No
	StAX API only	No

Filtering not in use

This feature has been tested with Woodstox 3.2.9 and Sun JDK 1.6 StAX implementation.

New for Camel 2.12.1

JaxbDataFormat now allows you to customize the XMLStreamWriter used to marshal the stream to XML. Using this configuration, you can add your own stream writer to completely remove, escape, or replace non-xml characters.

```
java JaxbDataFormat customWriterFormat = new JaxbDataFormat("org.apache.camel.foo.bar");
customWriterFormat.setXmlStreamWriterWrapper(new TestXmlStreamWriter());
```

The following example shows using the Spring DSL and also enabling Camel's NonXML filtering:

```
xml:bean id="testXmlStreamWriterWrapper" class="org.apache.camel.jaxb.TestXmlStreamWriter"/> <jaxb
filterNonXmlChars="true" contextPath="org.apache.camel.foo.bar" xmlStreamWriterWrapper="#testXmlStreamWriterWrapper" />
```

Working with the ObjectFactory

If you use JJC to create the java class from the schema, you will get an ObjectFactory for you JAXB context. Since the ObjectFactory uses JAXBELEMENT to hold the reference of the schema and element instance value, jaxbDataformat will ignore the JAXBELEMENT by default and you will get the element instance value instead of the JAXBELEMENT object form the unmarshaled message body.

If you want to get the JAXBELEMENT object form the unmarshaled message body, you need to set the JaxbDataFormat object's ignoreJAXBELEMENT property to be false.

Setting encoding

You can set the **encoding** option to use when marshalling. Its the `Marshaller.JAXB_ENCODING` encoding property on the JAXB Marshaller.

You can setup which encoding to use when you declare the JAXB data format. You can also provide the encoding in the Exchange property `Exchange.CHARSET_NAME`. This property will overrule the encoding set on the JAXB data format.

In this Spring DSL we have defined to use `iso-8859-1` as the encoding:

```
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-
jaxb/src/test/resources/org/apache/camel/example/springDataFormatWithEncoding.xml}
```

Controlling namespace prefix mapping

Available as of Camel 2.11

When marshalling using JAXB or SOAP then the JAXB implementation will automatic assign namespace prefixes, such as ns2, ns3, ns4 etc. To control this mapping, Camel allows you to refer to a map which contains the desired mapping.

Notice this requires having JAXB-RI 2.1 or better (from SUN) on the classpath, as the mapping functionality is dependent on the implementation of JAXB, whether its supported.

For example in Spring XML we can define a Map with the mapping. In the mapping file below, we map SOAP to use soap as prefix. While our custom namespace "`http://www.mycompany.com/foo/2`" is not using any prefix.

```
xml <util:map id="myMap"> <entry key="http://www.w3.org/2003/05/soap-envelope" value="soap"/> <!-- we dont want any prefix
for our namespace --> <entry key="http://www.mycompany.com/foo/2" value="" /> </util:map>
```

To use this in JAXB or SOAP you refer to this map, using the `namespacePrefixRef` attribute as shown below. Then Camel will lookup in the Registry a `java.util.Map` with the id "myMap", which was what we defined above.

```
xml <marshal> <soapjaxb version="1.2" contextPath="com.mycompany.foo" namespacePrefixRef="myMap"/> </marshal>
```

Schema validation

Available as of Camel 2.11

The JAXB Data Format supports validation by marshalling and unmarshalling from/to XML. Your can use the prefix `classpath: file:*` or `*http:` to specify how the resource should be resolved. You can separate multiple schema files by using the `,` character.

Known issue

Camel 2.11.0 and 2.11.1 has a known issue by validation multiple Exchange's in parallel. See CAMEL-6630. This is fixed with Camel 2.11.2/2.12.0.

Using the Java DSL, you can configure it in the following way:

```
javaJaxbDataFormat jaxbDataFormat = new JaxbDataFormat();
jaxbDataFormat.setContextPath(Person.class.getPackage().getName());
jaxbDataFormat.setSchema("classpath:person.xsd,classpath:address.xsd");
```

You can do the same using the XML DSL:

```
xml:marshal> <jaxb id="jaxb" schema="classpath:person.xsd,classpath:address.xsd"/> </marshal>
```

Camel will create and pool the underling SchemaFactory instances on the fly, because the SchemaFactory shipped with the JDK is not thread safe.

However, if you have a SchemaFactory implementation which is thread safe, you can configure the JAXB data format to use this one:

```
javaJaxbDataFormat jaxbDataFormat = new JaxbDataFormat(); jaxbDataFormat.setSchemaFactory(threadSafeSchemaFactory);
```

Schema Location

Available as of Camel 2.14

The JAXB Data Format supports to specify the SchemaLocation when marshaling the XML.

Using the Java DSL, you can configure it in the following way:

```
javaJaxbDataFormat jaxbDataFormat = new JaxbDataFormat();
jaxbDataFormat.setContextPath(Person.class.getPackage().getName());
jaxbDataFormat.setSchemaLocation("schema/person.xsd");
```

You can do the same using the XML DSL:

```
xml<marshal> <jaxb id="jaxb" schemaLocation="schema/person.xsd"/> </marshal>
```

Marshal data that is already XML

Available as of Camel 2.14.1

The JAXB marshaller requires that the message body is JAXB compatible, eg its a JAXBElement, eg a java instance that has JAXB annotations, or extend JAXBElement. There can be situations where the message body is already in XML, eg from a String type. There is a new option `mustBeJAXBElement` you can set to false, to relax this check, so the JAXB marshaller only attempts to marshal JAXBElements (`javax.xml.bind.JAXBIntrospector#isElement` returns true). And in those situations the marshaller fallbacks to marshal the message body as-is.

XmIRootElement objects

Available as of Camel 2.17.2

The JAXB Data Format option `objectFactory` has a default value equals to false. This is related to a performance degrading. For more information look at the issue CAMEL-10043

For the marshalling of non-XmIRootElement JAXB objects you'll need to call `JaxbDataFormat#setObjectFactory(true)`

Dependencies

To use JAXB in your camel routes you need to add the a dependency on **camel-jaxb** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-jaxb</artifactId> <version>x.x.x</version>
</dependency>
```

XMLBEANS

XmlBeans is a Data Format which uses the XmlBeans library to unmarshal an XML payload into Java objects or to marshal Java objects into an XML payload.

```
from("activemq:My-Queue").
unmarshal().xmlBeans().
to("mqseries:Another-Queue");
```

Dependencies

To use XmlBeans in your camel routes you need to add the dependency on **camel-xmlbeans** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

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

XSTREAM

XStream is a Data Format which uses the XStream library to marshal and unmarshal Java objects to and from XML.

To use XStream in your camel routes you need to add the a dependency on **camel-xstream** which implements this data format.

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

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

Using the Java DSL

```
// lets turn Object messages into XML then send to MQSeries from("activemq:My.Queue"). marshal().xstream().  
to("mqseries:Another.Queue");  
If you would like to configure the xstream instance used by the Camel for the message transformation, you can simply pass a  
reference to that instance on the DSL level.  
XStream xStream = new XStream(); xStream.aliasField("money", PurchaseOrder.class, "cash"); // new Added setModel option  
since Camel 2.14 xStream.setModel("NO_REFERENCES"); ... from("direct:marshal"). marshal(new  
XStreamDataFormat(xStream)). to("mock:marshaled");
```

XMLInputFactory and XMLOutputFactory

The XStream library uses the javax.xml.stream.XMLInputFactory and javax.xml.stream.XMLOutputFactory, you can control which implementation of this factory should be used.

The Factory is discovered using this algorithm:

1. Use the javax.xml.stream.XMLInputFactory , javax.xml.stream.XMLOutputFactory system property.
2. Use the lib/xml/stream.properties file in the JRE_HOME directory.
3. Use the Services API, if available, to determine the classname by looking in the META-INF/services/javax.xml.stream.XMLInputFactory, META-INF/services/javax.xml.stream.XMLOutputFactory files in jars available to the JRE.
4. Use the platform default XMLInputFactory,XMLOutputFactory instance.

How to set the XML encoding in Xstream DataFormat?

From Camel 2.2.0, you can set the encoding of XML in Xstream DataFormat by setting the Exchange's property with the key Exchange.CHARSET_NAME, or setting the encoding property on Xstream from DSL or Spring config.

```
from("activemq:My.Queue"). marshal().xstream("UTF-8"). to("mqseries:Another.Queue");  
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-xstream/src/test/resources/org/apache/camel/dataformat/xstream/SpringMarshalListTest.xml}
```

Setting the type permissions of Xstream DataFormat

In Camel, one can always use its own processing step in the route to filter and block certain XML documents to be routed to the XStream's unmarshall step. From Camel 2.16.1, 2.15.5, you can set XStream's type permissions to automatically allow or deny the instantiation of certain types.

The default type permissions setting used by Camel denies all types except for those from java.lang and java.util packages. This setting can be changed by setting System property org.apache.camel.xstream.permissions. Its value is a string of comma-separated permission terms, each representing a type being allowed or denied, depending on whether the term is prefixed with '+' (note '+' may be omitted) or with '-', respectively.

Each term may contain a wildcard character '*'. For example, value "-*java.lang.*java.util.*" indicates denying all types except for java.lang.* and java.util.* classes. Setting this value to an empty string "" reverts to the default XStream's type permissions handling which denies certain blacklisted classes and allow others.

The type permissions setting can be extended at an individual XStream DataFormat instance by setting its type permissions property.

```
<dataFormats> <xstream id="xstream-default" permissions="org.apache.camel.samples.xstream.*"/> ...
```

CSV

The CSV Data Format uses Apache Commons CSV to handle CSV payloads (Comma Separated Values) such as those exported/imported by Excel.

As of Camel 2.15.0, it now uses the Apache Commons CSV 1.1 which is based on a completely different set of options.

Available options until Camel 2.15

Option	Type	Description
config	CSVConfig	Can be used to set a custom csvConfig object.
strategy	CSVStrategy	Can be used to set a custom CSVstrategy; the default is csvStrategy.DEFAULT_STRATEGY.
autogenColumns	boolean	Whether or not columns are auto-generated in the resulting CSV. The default value is true; subsequent messages use the previously created columns with new fields being added at the end of the line.
delimiter	String	Camel 2.4: The column delimiter to use; the default value is ",".
skipFirstLine	boolean	Camel 2.10: Whether or not to skip the first line of CSV input when unmarshalling (e.g. if the content has headers on the first line); the default value is false.
lazyLoad	boolean	Camel 2.12.2: Whether or not to Sequential access CSV input through an iterator which could avoid OOM exception when processing huge CSV file; the default value is false

useMaps	boolean	Camel 2.13: Whether to use <code>List<Map></code> when unmarshalling instead of <code>List<List></code> .
Available options as of Camel 2.15		
Option	Type	Description
format	CSVFormat	The reference format to use, it will be updated with the other format options, the default value is <code>CSVFormat.DEFAULT</code>
commentMarkerDisabled	boolean	Disables the comment marker of the reference format. This option is <code>false</code> by default.
commentMarker	Character	Overrides the comment marker of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>null</code> for <code>CSVFormat.DEFAULT</code> .
delimiter	Character	Overrides the delimiter of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>,</code> for <code>CSVFormat.DEFAULT</code> .
escapeddisabled	boolean	Disables the escape character of the reference format. This option is <code>false</code> by default.
escape	Character	Overrides the escape character of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>null</code> for <code>CSVFormat.DEFAULT</code> .
headerDisabled	boolean	Disables the header of the reference format. This option is <code>false</code> by default.
header	String[]	Overrides the header of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>null</code> for <code>CSVFormat.DEFAULT</code> . In the XML DSL, this option is configured using children <code><header></code> tags: <pre><csv> <header>orderId</header> <header>amount</header> </csv></pre>
allowMissingColumnNames	Boolean	Overrides the missing column names behavior of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>false</code> for <code>CSVFormat.DEFAULT</code> .
ignoreEmptyLines	Boolean	Overrides the empty line behavior of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>true</code> for <code>CSVFormat.DEFAULT</code> .
ignoreSurroundingspaces	Boolean	Overrides the surrounding spaces behavior of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>false</code> for <code>CSVFormat.DEFAULT</code> .
nullStringdisabled	boolean	Disables the null string representation of the reference format. This option is <code>false</code> by default.
nullString	String	Overrides the null string representation of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>null</code> for <code>CSVFormat.DEFAULT</code> .
quoteDisabled	boolean	Disables the quote of the reference format. This option is <code>false</code> by default.
quote	Character	Overrides the quote symbol of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>"</code> (double quote) for <code>CSVFormat.DEFAULT</code> .
quoteMode	QuoteMode	Overrides the quote mode of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>null</code> for <code>CSVFormat.DEFAULT</code> .
recordSeparatorDisabled	boolean	Disables the record separator of the reference format. This option is <code>false</code> by default.
recordseparator	String	Overrides the record separator of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>\r\n</code> (CRLF) for <code>CSVFormat.DEFAULT</code> .

skipHeaderRecord	Boolean	Overrides the header record behavior of the reference format. This option is <code>null</code> by default. When <code>null</code> it keeps the value of the reference format which is <code>false</code> for <code>CSVFormat.DEFAULT</code> .
lazyLoad	boolean	Whether the unmarshalling should produce an iterator that reads the lines on the fly or if all the lines must be read at one. This option is <code>false</code> by default.
useMaps	boolean	Whether the unmarshalling should produce maps for the lines values instead of lists. It requires to have header (either defined or collected). This option is <code>false</code> by default.
recordConverter	CsvRecordConverter	Sets the record converter to use. If defines the <code>useMaps</code> option is disabled. This option is <code>null</code> by default.

Marshalling a Map to CSV

The component allows you to marshal a Java Map (or any other message type that can be converted in a Map) into a CSV payload.

Considering the following body

```
Map<String, Object> body = new LinkedHashMap<>();
body.put("foo", "abc");
body.put("bar", 123);
```

and this Java route definition

```
from("direct:start")
    .marshal().csv()
    .to("mock:result");
```

or this XML route definition

```
<route>
    <from uri="direct:start" />
    <marshal>
        <csv />
    </marshal>
    <to uri="mock:result" />
</route>
```

then it will produce

```
abc,123
```

Unmarshalling a CSV message into a Java List

Unmarshalling will transform a CSV message into a Java List with CSV file lines (containing another List with all the field values).

An example: we have a CSV file with names of persons, their IQ and their current activity.

```
Jack Dalton, 115, mad at Averell
Joe Dalton, 105, calming Joe
William Dalton, 105, keeping Joe from killing Averell
Averell Dalton, 80, playing with Rantanplan
Lucky Luke, 120, capturing the Daltons
```

We can now use the CSV component to unmarshal this file:

```
from("file:src/test/resources/?fileName=daltons.csv&noop=true")
    .unmarshal().csv()
    .to("mock:daltons");
```

The resulting message will contain a `List<List<String>>` like...

```
List<List<String>> data = (List<List<String>>) exchange.getIn().getBody();
for (List<String> line : data) {
    LOG.debug(String.format("%s has an IQ of %s and is currently %s", line.get(0), line.get(1), line.get(2)));}
```

Marshalling a List<Map> to CSV

Available as of Camel 2.1

If you have multiple rows of data you want to be marshalled into CSV format you can now store the message payload as a `List<Map<String, Object>>` object where the list contains a Map for each row.

File Poller of CSV, then unmarshaling

Given a bean which can handle the incoming data...

```
Listing 1. MyCsvHandler.java

// Some comments here
public void doHandleCsvData(List<List<String>> csvData)
{
    // do magic here
}
```

... your route then looks as follows

```
<route>
    <!-- poll every 10 seconds -->
    <from uri="file:///some/path/to/pickup/csvfiles?delete=true&consumer.delay=10000" />
    <unmarshal><csv /></unmarshal>
    <to uri="bean:myCsvHandler?method=doHandleCsvData" />
</route>
```

Marshaling with a pipe as delimiter

Considering the following body

```
Map<String, Object> body = new LinkedHashMap<>();
body.put("foo", "abc");
body.put("bar", 123);

// Camel version < 2.15
CsvDataFormat oldCSV = new CsvDataFormat();
oldCSV.setDelimiter("|");
from("direct:start")
    .marshal(oldCSV)
    .to("mock:result")
// Camel version >= 2.15
from("direct:start")
    .marshal(new CsvDataFormat().setDelimiter(&#39;|'))
    .to("mock:result")
```

and this Java route definition

```
<route>
    <from uri="direct:start" />
    <marshal>
        <csv delimiter="|" />
    </marshal>
    <to uri="mock:result" />
</route>
```

or this XML route definition

```
abc|123
```

Using autogenColumns, configRef and strategyRef attributes inside XML DSL

Available as of Camel 2.9.2 / 2.10 and deleted for Camel 2.15

You can customize the CSV Data Format to make use of your own csvConfig and/or csvStrategy. Also note that the default value of the autogenColumns option is true. The following example should illustrate this customization.

```
<route>
    <from uri="direct:start" />
    <marshal>
        <!-- make use of a strategy other than the default one which is &#39;org.apache.commons.csv.CSVStrategy.DEFAULT_STRATEGY' -->
        <csv autogenColumns="false" delimiter="|" configRef="csvConfig" strategyRef="excelStrategy" />
    </marshal>
    <convertBodyTo type="java.lang.String" />
    <to uri="mock:result" />
</route>

<bean id="csvConfig" class="org.apache.commons.csv.writer.CSVConfig">
    <property name="fields">
        <list>
            <bean class="org.apache.commons.csv.writer.CSVField">
                <property name="name" value="orderId" />
            </bean>
            <bean class="org.apache.commons.csv.writer.CSVField">
                <property name="name" value="amount" />
            </bean>
        </list>
    </property>
</bean>

<bean id="excelStrategy" class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
    <property name="staticField" value="org.apache.commons.csv.CSVStrategy.EXCEL_STRATEGY" />

```

```
</bean>
```

Using skipFirstLine option while unmarshaling

Available as of Camel 2.10 and deleted for Camel 2.15

You can instruct the CSV Data Format to skip the first line which contains the CSV headers. Using the Spring/XML DSL:

```
<route>
  <from uri="direct:start" />
  <unmarshal>
    <csv skipFirstLine="true" />
  </unmarshal>
  <to uri="bean:myCsvHandler?method=doHandleCsv" />
</route>
```

Or the Java DSL:

```
CsvDataFormat csv = new CsvDataFormat();
csv.setSkipFirstLine(true);

from("direct:start")
.unmarshal(csv)
.to("bean:myCsvHandler?method=doHandleCsv");
```

Unmarshaling with a pipe as delimiter

Using the Spring/XML DSL:

```
<route>
  <from uri="direct:start" />
  <unmarshal>
    <csv delimiter="|" />
  </unmarshal>
  <to uri="bean:myCsvHandler?method=doHandleCsv" />
</route>
```

Or the Java DSL:

```
CsvDataFormat csv = new CsvDataFormat();
CSVStrategy strategy = CSVStrategy.DEFAULT_STRATEGY;
strategy.setDelimiter(';' );
csv.setStrategy(strategy);

from("direct:start")
.unmarshal(csv)
.to("bean:myCsvHandler?method=doHandleCsv");

CsvDataFormat csv = new CsvDataFormat();
csv.setDelimiter('|');

from("direct:start")
.unmarshal(csv)
.to("bean:myCsvHandler?method=doHandleCsv");

CsvDataFormat csv = new CsvDataFormat();
CSVConfig csvConfig = new CSVConfig();
csvConfig.setDelimiter(';');
csv.setConfig(csvConfig);

from("direct:start")
.unmarshal(csv)
.to("bean:myCsvHandler?method=doHandleCsv");
```

Issue in CSVConfig

It looks like that

```
CSVConfig csvConfig = new CSVConfig();
csvConfig.setDelimiter(';' );
```

doesn't work. You have to set the delimiter as a String!

Dependencies

To use CSV in your Camel routes you need to add a dependency on **camel-csv**, which implements this data format.

If you use Maven you can just add the following to your pom.xml, substituting the version number for the latest and greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-csv</artifactId>
  <version>x.x.x</version>
```

```
</dependency>
```

The String Data Format is a textual based format that supports encoding.

Options

Option	Default	Description
charset	null	To use a specific charset for encoding. If not provided Camel will use the JVM default charset.

Marshal

In this example we marshal the file content to String object in UTF-8 encoding.

```
from("file://data.csv").marshal().string("UTF-8").to("jms://myqueue");
```

Unmarshal

In this example we unmarshal the payload from the JMS queue to a String object using UTF-8 encoding, before its processed by the newOrder processor.

```
from("jms://queue/order").unmarshal().string("UTF-8").processRef("newOrder");
```

Dependencies

This data format is provided in **camel-core** so no additional dependencies is needed.

HL7 DataFormat

The HL7 component ships with a HL7 data format that can be used to marshal or unmarshal HL7 model objects.

- `marshal` = from Message to byte stream (can be used when responding using the HL7 MLLP codec)
- `unmarshal` = from byte stream to Message (can be used when receiving streamed data from the HL7 MLLP)

To use the data format, simply instantiate an instance and invoke the marshal or unmarshal operation in the route builder:

```
DataFormat h17 = new HL7DataFormat();
...
from("direct:h17in").marshal(h17).to("jms:queue:h17out");
```

In the sample above, the HL7 is marshalled from a HAPI Message object to a byte stream and put on a JMS queue.

The next example is the opposite:

```
DataFormat h17 = new HL7DataFormat();
...
from("jms:queue:h17out").unmarshal(h17).to("patientLookupService");
```

Here we unmarshal the byte stream into a HAPI Message object that is passed to our patient lookup service.

Serializable messages

As of HAPI 2.0 (used by **Camel 2.11**), the HL7v2 model classes are fully serializable. So you can put HL7v2 messages directly into a JMS queue (i.e. without calling `marshal()` and read them again directly from the queue (i.e. without calling `unmarshal()`). Segment separators

As of **Camel 2.11**, `unmarshal` does not automatically fix segment separators anymore by converting `\n` to `\r`. If you need this conversion, `org.apache.camel.component.hl7.HL7#convertLFToCR` provides a handy Expression for this purpose.

Charset

As of **Camel 2.14.1**, both `marshal` and `unmarshal` evaluate the charset provided in the field `MSH-18`. If this field is empty, by default the charset contained in the corresponding Camel charset property/header is assumed. You can even change this default behavior by overriding the `guessCharsetName` method when inheriting from the `HL7DataFormat` class.

There is a shorthand syntax in Camel for well-known data formats that are commonly used.
Then you don't need to create an instance of the `HL7DataFormat` object:

```
from("direct:h17in").marshal().h17().to("jms:queue:h17out");
from("jms:queue:h17out").unmarshal().h17().to("patientLookupService");
```

EDI DATAFORMAT

We encourage end users to look at the Smooks which supports EDI and Camel natively.

FLATPACK DATAFORMAT

The Flatpack component ships with the Flatpack data format that can be used to format between fixed width or delimited text messages to a List of rows as Map.

- marshal = from List<Map<String, Object>> to OutputStream (can be converted to String)
- unmarshal = from java.io.InputStream (such as a File or String) to a java.util.List as an org.apache.camel.component.flatpack.DataSetList instance.
The result of the operation will contain all the data. If you need to process each row one by one you can split the exchange, using Splitter.

Notice: The Flatpack library does currently not support header and trailers for the marshal operation.

Options

The data format has the following options:

Option	Default	Description
definition	null	The flatpack pzmap configuration file. Can be omitted in simpler situations, but its preferred to use the pzmap.
fixed	false	Delimited or fixed.
ignoreFirstRecord	true	Whether the first line is ignored for delimited files (for the column headers).
textQualifier	"	If the text is qualified with a char such as "
delimiter	,	The delimiter char (could be ; , or similar)
parserFactory	null	Uses the default Flatpack parser factory.
allowShortLines	false	Camel 2.9.7 and 2.10.5 onwards: Allows for lines to be shorter than expected and ignores the extra characters.
ignoreExtraColumns	false	Camel 2.9.7 and 2.10.5 onwards: Allows for lines to be longer than expected and ignores the extra characters.

Usage

To use the data format, simply instantiate an instance and invoke the marshal or unmarshal operation in the route builder:

```
FlatpackDataFormat fp = new FlatpackDataFormat();
fp.setDefinition(new ClassPathResource("INVENTORY-Delimited.pzmap.xml"));
...
from("file:order/in").unmarshal(df).to("seda:queue:neworder");
```

The sample above will read files from the order/in folder and unmarshal the input using the Flatpack configuration file INVENTORY-Delimited.pzmap.xml that configures the structure of the files. The result is a DataSetList object we store on the SEDA queue.

```
FlatpackDataFormat df = new FlatpackDataFormat();
df.setDefinition(new ClassPathResource("PEOPLE-FixedLength.pzmap.xml"));
df.setFixed(true);
df.setIgnoreFirstRecord(false);

from("seda:people").marshal(df).convertBodyTo(String.class).to("jms:queue:people");
```

In the code above we marshal the data from a Object representation as a List of rows as Maps. The rows as Map contains the column name as the key, and the the corresponding value. This structure can be created in Java code from e.g. a processor. We marshal the data according to the Flatpack format and convert the result as a String object and store it on a JMS queue.

Dependencies

To use Flatpack in your camel routes you need to add the a dependency on **camel-flatpack** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-flatpack</artifactId>
  <version>x.x.x</version>
</dependency>
```

JSON

JSON is a Data Format to marshal and unmarshal Java objects to and from JSON.

For JSON to object marshalling, Camel provides integration with three popular JSON libraries:

- The XStream library and Jettsion
- The Jackson library
- **Camel 2.10: The GSon library**

Every library requires adding the special camel component (see "Dependency.." paragraphs further down). By default Camel uses the XStream library.

Direct, bi-directional JSON <=> XML conversions

As of Camel 2.10, Camel supports direct, bi-directional JSON <=> XML conversions via the camel-xmljson data format, which is documented separately.

Using JSON Data Format With the xstream Library

```
java// Let's turn Object messages into JSON then send to MQSeries from("activemq:My.Queue") .marshal().json()
.to("mqseries:Another.Queue");
```

Using JSON Data Format With the Jackson Library

```
java// Let's turn Object messages into JSON then send to MQSeries from("activemq:My.Queue")
.marshal().json(JsonLibrary.Jackson) .to("mqseries:Another.Queue");
```

Using JSON Data Format With the GSon Library

```
java// Let's turn Object messages into JSON then send to MQSeries from("activemq:My.Queue")
.marshal().json(JsonLibrary.Gson) .to("mqseries:Another.Queue");
```

Using JSON in Spring DSL

When using Data Format in Spring DSL you need to declare the data formats first. This is done in the `dataFormats` XML tag. `<!-- Here we define a Json data format with the id jack and that it should use the TestPojo as the class type when doing unmarshal. The unmarshalType is optional, if not provided Camel will use a Map as the type. --> <json id="jack" library="Jackson" unmarshalType="org.apache.camel.component.jackson.TestPojo"/>`

And then you can refer to this id in the route:

```
xml<route> <from uri="direct:back"/> <unmarshal ref="jack"/> <to uri="mock:reverse"/> </route>
```

Excluding POJO Fields From Marshalling

As of Camel 2.10

When marshaling a POJO to JSON you might want to exclude certain fields from the JSON output. With Jackson you can use JSON views to accomplish this.

First create one or more marker classes:`{snippet:id=marker|lang=java|url=camel/trunk/components/camel-jackson/src/test/java/org/apache/camel/component/jackson/Views.java}`Second, use the marker classes with the `@JsonView` annotation to include/exclude certain fields. The annotation also works on getters:
`{snippet:id=jsonview|lang=java|url=camel/trunk/components/camel-jackson/src/test/java/org/apache/camel/component/jackson/TestPojoView.java}`Finally, use the Camel `JacksonDataFormat` to marshal the above POJO to JSON.`{snippet:id=format|lang=java|url=camel/trunk/components/camel-jackson/src/test/java/org/apache/camel/component/jackson/JacksonMarshalViewTest.java}`**Note:** the height field is missing in the resulting JSON.

`{"age":30, "weight":70}`

The GSon library supports a similar feature through the notion of ExclusionStrategies:

```
{snippet:id=strategy|lang=java|url=camel/trunk/components/camel-gson/src/test/java/org/apache/camel/component/gson/GsonMarshalExclusionTest.java}The GsonDataFormat accepts an ExclusionStrategy in its constructor:{snippet:id=format|lang=java|url=camel/trunk/components/camel-gson/src/test/java/org/apache/camel/component/gson/GsonMarshalExclusionTest.java}The line above will exclude fields annotated with @ExcludeAge when marshaling to JSON.
```

Configuring Field Naming Policy

Available as of Camel 2.11

The GSon library supports specifying policies and strategies for mapping from JSON to POJO fields. A common naming convention is to map JSON fields using lower case with underscores.

We may have this JSON string

```
{ "id" : 123, "first_name" : "Donald" "last_name" : "Duck" }
```

Which we want to map to a POJO that has getter/setters as:

```
javaPersonPojo.javaPersonPojo.javapublic class PersonPojo { private int id; private String firstName; private String lastName; public int getId() { return id; } public void setId(int id) { this.id = id; } public String getFirstName() { return firstName; } public void setFirstName(String firstName) { this.firstName = firstName; } public String getLastName() { return lastName; } public void setLastName(String lastName) { this.lastName = lastName; }}
```

Then we can configure the `org.apache.camel.component.gson.GsonDataFormat` in a Spring XML file as shown below. Notice we use `fieldNamingPolicy` property to set the field mapping. This property is an enum from Gson `com.google.gson.FieldNamingPolicy` which has a number of predefined mappings.

If you need full control you can use the property `FieldNamingStrategy` and implement a custom `com.google.gson.FieldNamingStrategy` where you can control the mapping.

Configuring `GsonDataFormat` in Spring XML file

```
<!-- define the gson data format, where we configure the data format using the properties --> <bean id="gson" class="org.apache.camel.component.gson.GsonDataFormat"> <!-- we want to unmarshal to person pojo --> <property name="unmarshalType" value="org.apache.camel.component.gson.PersonPojo"/> <!-- we want to map fields to use lower case and underscores --> <property name="fieldNamingPolicy" value="LOWER_CASE_WITH_UNDERSCORES"/> </bean>
```

And use it in Camel routes by referring to its bean id as shown:

```
Using gson from Camel routesxmlUsing gson from Camel Routes<camelContext xmlns="http://camel.apache.org/schema/spring"> <route> <from uri="direct:inPojo"/> <marshal ref="gson"/> </route> <route> <from uri="direct:backPojo"/> <unmarshal ref="gson"/> </route> </camelContext>
```

Include/Exclude Fields Using the `jsonview` Attribute With `JacksonDataFormat`

Available as of Camel 2.12

As an example of using this attribute you can instead of:

```
javaJacksonDataFormat ageViewFormat = new JacksonDataFormat(TestPojoView.class, Views.Age.class); from("direct:inPojoAgeView").marshal(ageViewFormat); Directly specify your JSON view inside the Java DSL as: javafrom("direct:inPojoAgeView") .marshal(TestPojoView.class, Views.Age.class); And the same in XML DSL: xml<from uri="direct:inPojoAgeView"/> <marshal> <json library="Jackson" unmarshalTypeName="org.apache.camel.component.jackson.TestPojoView" jsonView="org.apache.camel.component.jackson.Views$Age"/> </marshal>
```

Setting Serialization Include Option for Jackson Marshal

Available as of Camel 2.13.3/2.14

If you want to marshal a POJO to JSON, and the POJO has some fields with null values. And you want to skip these null values, then you need to set either an annotation on the POJO,

```
java@JsonInclude(JsonInclude.Include.NON_NULL) public class MyPojo { / ... }
```

But this requires you to include that annotation in your POJO source code. You can also configure the Camel `JacksonDataFormat` to set the include option, as shown below:

```
javaJacksonDataFormat format = new JacksonDataFormat(); format.setInclude("NON_NULL"); Or from XML DSL you configure this as xml<dataFormats> <json id="json" library="Jackson" include="NON_NULL"/> </dataFormats>
```

Unmarshaling from JSON to POJO with Dynamic Class Name

Available as of Camel 2.14

If you use Jackson to unmarshal JSON to POJO, then you can now specify a header in the message that indicate which class name to unmarshal to. The header has key `CamelJacksonUnmarshalType` if that header is present in the message, then Jackson will use that as FQN for the POJO class to unmarshal the JSON payload as. Notice that behavior is enabled out of the box from **Camel 2.14**.

For JMS end users there is the `JMSType` header from the JMS spec that indicates that also. To enable support for `JMSType` you would need to turn that on, on the Jackson data format as shown:

```
javaJacksonDataFormat format = new JacksonDataFormat(); format.setAllowJmsType(true); Or from XML DSL you configure this as xml<dataFormats> <json id="json" library="Jackson" allowJmsType="true"/> </dataFormats>
```

Unmarshaling From JSON to `List<Map>` or `List<pojo>`

Available as of Camel 2.14

If you are using Jackson to unmarshal JSON to a list of map/POJO, you can now specify this by setting `useList="true"` or use the `org.apache.camel.component.jackson.ListJacksonDataFormat`.

For example, with Java you can do as shown below:

```
javaJacksonDataFormat format = new ListJacksonDataFormat(); // or JacksonDataFormat format = new JacksonDataFormat(); format.useList(); // and you can specify the pojo class type also format.setUnmarshalType(MyPojo.class); And if you use XML DSL then you configure to use list using useList attribute as shown below:
```

```
xml<dataFormats> <json id="json" library="Jackson" useList="true"/> </dataFormats>
And you can specify the pojo type also
xml<dataFormats> <json id="json" library="Jackson" useList="true" unmarshalTypeName="com.foo.MyPojo"/> </dataFormats>
```

Using Custom Jackson's ObjectMapper

Available from Camel 2.17

You can use custom Jackson objectMapper instance, can be configured as shown below.

```
xml<dataFormats> <json id="json" library="Jackson" objectMapper="myMapper"/> </dataFormats>
Where myMapper is the id of the custom instance that Camel will lookup in the Registry.
```

Using Custom Jackson Modules

Available as of Camel 2.15

You can use custom Jackson modules by specifying the class names of those using the moduleClassNames option as shown below.

```
xml<dataFormats> <json id="json" library="Jackson" useList="true" unmarshalTypeName="com.foo.MyPojo"
moduleClassNames="com.foo.MyModule,com.foo.MyOtherModule"/> </dataFormats>
When using moduleClassNames then the custom Jackson modules are not configured, by created using default constructor and used as-is. If a custom module needs any custom configuration, then an instance of the module can be created and configured, and then use moduleRefs to refer to the module as shown below:
xml<bean id="myJacksonModule" class="com.foo.MyModule"> ... // configure the module as you want </bean> <dataFormats>
<json id="json" library="Jackson" useList="true" unmarshalTypeName="com.foo.MyPojo" moduleRefs="myJacksonModule"/>
</dataFormats>
Multiple modules can be specified separated by comma, such as moduleRefs="myJacksonModule,myOtherModule".
```

Enabling or Disable Features Using Jackson

Available as of Camel 2.15

Jackson has a number of features you can enable or disable, which its objectMapper uses. For example to disable failing on unknown properties when marshaling, you can configure this using the disableFeatures:

```
xml<dataFormats> <json id="json" library="Jackson" unmarshalTypeName="com.foo.MyPojo"
disableFeatures="FAIL_ON_UNKNOWN_PROPERTIES"/> </dataFormats>
```

You can disable multiple features by separating the values using comma. The values for the features must be the name of the enums from Jackson from the following enum classes

- com.fasterxml.jackson.databind.SerializationFeature
- com.fasterxml.jackson.databind.DeserializationFeature
- com.fasterxml.jackson.databind.MapperFeature

To enable a feature use the enableFeatures options instead.

```
From Java code you can use the type safe methods from camel-jackson module:
javaJacksonDataFormat df = new JacksonDataFormat(MyPojo.class);
df.disableFeature(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);
df.disableFeature(DeserializationFeature.FAIL_ON_NULL_FOR_PRIMITIVES);
```

Converting Maps to POJO Using Jackson

Available since **Camel 2.16**. Jackson ObjectMapper can be used to convert maps to POJO objects. Jackson component comes with the data converter that can be used to convert java.util.Map instance to non-String, non-primitive and non-Number objects.

```
javaMap<String, Object> invoiceData = new HashMap<String, Object>(); invoiceData.put("netValue", 500);
producerTemplate.sendBody("direct:mapToInvoice", invoiceData); ... // Later in the processor Invoice invoice =
exchange.getIn().getBody(Invoice.class);
If there is a single objectMapper instance available in the Camel registry, it will used by the converter to perform the conversion.
Otherwise the default mapper will be used.
```

Formatted JSON marshalling (pretty-printing)

Available as of Camel 2.16

Using the prettyPrint option one can output a well formatted JSON while marshaling:

```
xml<dataFormats> <json id="xstream" prettyPrint="true"/> <json id="jackson" prettyPrint="true" library="Jackson"/> <json
id="gson" prettyPrint="true" library="Gson"/> </dataFormats>
```

And in Java DSL:

```
javafrom("direct:inPretty").marshal().json(true); from("direct:inPretty").marshal().json(JsonLibrary.Jackson, true);
from("direct:inPretty").marshal().json(JsonLibrary.Gson, true);
```

Please note that as of Camel 2.16 there are five different overloaded `json()` DSL methods which support the `prettyPrint` option in combination with other settings for `JsonLibrary`, `unmarshalType`, `jsonView` etc.

Integrating Jackson with Camel's TypeConverters

Available as of Camel 2.17

The `camel-jackson` module allows to integrate Jackson as a Type Converter in the Camel registry. This works in similar ways that `camel-jaxb` integrates with the type converter as well. However `camel-jackson` must be explicitly enabled, which is done by setting some options on the `CamelContext` properties, as shown below:

```
java// Enable Jackson JSON type converter. getContext().getProperties().put("CamelJacksonEnableTypeConverter", "true"); //  
Allow Jackson JSON to convert to pojo types also (by default Jackson only converts to String and other simple types).  
getContext().getProperties().put("CamelJacksonTypeConverterToPojo", "true");
```

The `camel-jackson` type converter integrates with JAXB which means you can annotate POJO class with JAXB annotations that Jackson can leverage.

Dependencies for XStream

To use JSON in your camel routes you need to add the dependency on `camel-xstream` which implements this data format.

If you use maven you could just add the following to your `pom.xml`, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-xstream</artifactId> <version>2.9.2</version>  
</dependency>
```

Dependencies for Jackson

To use JSON in your camel routes you need to add the dependency on `camel-jackson` which implements this data format.

If you use maven you could just add the following to your `pom.xml`, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-jackson</artifactId> <version>2.9.2</version>  
</dependency>
```

Dependencies for GSON

To use JSON in your camel routes you need to add the dependency on `camel-gson` which implements this data format.

If you use maven you could just add the following to your `pom.xml`, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-gson</artifactId> <version>2.10.0</version>  
</dependency> The Zip Data Format is a message compression and de-compression format. Messages marshalled using Zip compression can be unmarshalled using Zip decompression just prior to being consumed at the endpoint. The compression capability is quite useful when you deal with large XML and Text based payloads. It facilitates more optimal use of network bandwidth while incurring a small cost in order to compress and decompress payloads at the endpoint.
```

About using with Files

The Zip data format, does not (yet) have special support for files. Which means that when using big files, the entire file content is loaded into memory.

This is subject to change in the future, to allow a streaming based solution to have a low memory footprint.

Options

Option	Default	Description
<code>compressionLevel</code>	<code>null</code>	<p>To specify a specific compression Level use <code>java.util.zip.Deflater</code> settings. The possible settings are</p> <ul style="list-style-type: none"> - <code>Deflater.BEST_SPEED</code> - <code>Deflater.BEST_COMPRESSION</code> - <code>Deflater.DEFAULT_COMPRESSION</code> <p>If <code>compressionLevel</code> is not explicitly specified the <code>compressionLevel</code> employed is <code>Deflater.DEFAULT_COMPRESSION</code></p>

Marshal

In this example we marshal a regular text/XML payload to a compressed payload employing zip compression `Deflater.BEST_COMPRESSION` and send it an ActiveMQ queue called `MY_QUEUE`.

```
from("direct:start").marshal().zip(Deflater.BEST_COMPRESSION).to("activemq:queue:MY_QUEUE");
```

Alternatively if you would like to use the default setting you could send it as

```
from("direct:start").marshal().zip().to("activemq:queue:MY_QUEUE");
```

Unmarshal

In this example we unmarshal a zipped payload from an ActiveMQ queue called MY_QUEUE to its original format, and forward it for processing to the UnZippedMessageProcessor. Note that the compression Level employed during the marshalling should be identical to the one employed during unmarshalling to avoid errors.

```
from("activemq:queue:MY_QUEUE").unmarshal().zip().process(new UnZippedMessageProcessor());
```

Dependencies

This data format is provided in **camel-core** so no additional dependencies are needed.

TIDYMARKUP

TidyMarkup is a Data Format that uses the TagSoup to tidy up HTML. It can be used to parse ugly HTML and return it as pretty wellformed HTML.

Camel eats our own -dog food- soap

We had some issues in our pdf Manual where we had some strange symbols. So Jonathan used this data format to tidy up the wiki html pages that are used as base for rendering the pdf manuals. And then the mysterious symbols vanished.

TidyMarkup only supports the **unmarshal** operation as we really don't want to turn well formed HTML into ugly HTML 😊

Java DSL Example

An example where the consumer provides some HTML

```
from("file://site/inbox").unmarshal().tidyMarkup().to("file://site/blogs");
```

Spring XML Example

The following example shows how to use TidyMarkup to unmarshal using Spring

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
<route>
<from uri="file://site/inbox"/>
<unmarshal>
<tidyMarkup/>
</unmarshal>
<to uri="file://site/blogs"/>
</route>
</camelContext>
```

Dependencies

To use TidyMarkup in your camel routes you need to add the a dependency on **camel-tagsoup** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-tagsoup</artifactId>
<version>x.x.x</version>
</dependency>
```

BINDY

The goal of this component is to allow the parsing/binding of non-structured data (or to be more precise non-XML data) to/from Java Beans that have binding mappings defined with annotations. Using Bindy, you can bind data from sources such as :

- CSV records,
- Fixed-length records,
- FIX messages,

- or almost any other non-structured data

to one or many Plain Old Java Object (POJO). Bindy converts the data according to the type of the java property. POJOs can be linked together with one-to-many relationships available in some cases. Moreover, for data type like Date, Double, Float, Integer, Short, Long and BigDecimal, you can provide the pattern to apply during the formatting of the property.

For the BigDecimal numbers, you can also define the precision and the decimal or grouping separators.

Type	Format Type	Pattern example	Link
Date	DateFormat	"dd-MM-yyyy"	http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html
Decimal*	DecimalFormat	"##.###.###"	http://java.sun.com/j2se/1.5.0/docs/api/java/text/DecimalFormat.html

Decimal* = Double, Integer, Float, Short, Long

Format supported

This first release only support comma separated values fields and key value pair fields (e.g. : FIX messages).

To work with camel-bindy, you must first define your model in a package (e.g. com.acme.model) and for each model class (e.g. Order, Client, Instrument, ...) add the required annotations (described hereafter) to the Class or field.

Multiple models

If you use multiple models, each model has to be placed in it's own package to prevent unpredictable results.

From Camel 2.16 onwards this is no longer the case, as you can safely have multiple models in the same package, as you configure bindy using class names instead of package names now.

ANNOTATIONS

The annotations created allow to map different concept of your model to the POJO like :

- Type of record (csv, key value pair (e.g. FIX message), fixed length ...),
- Link (to link object in another object),
- DataField and their properties (int, type, ...),
- KeyValuePairField (for key = value format like we have in FIX financial messages),
- Section (to identify header, body and footer section),
- OneToMany

This section will describe them :

1. CsvRecord

The CsvRecord annotation is used to identified the root class of the model. It represents a record = a line of a CSV file and can be linked to several children model classes.

Annotation name	Record type	Level
CsvRecord	csv	Class
Parameter name	type	Info
separator	string	mandatory - can be ',' or ';' or 'anything'. This value is interpreted as a regular expression. If you want to use a sign which has a special meaning in regular expressions, e.g. the ';' sign, than you have to mask it, like '
skipFirstLine	boolean	optional - default value = false - allow to skip the first line of the CSV file
crlf	string	optional - possible values = WINDOWS,UNIX,MAC, or custom; default value = WINDOWS - allow to define the carriage return character to use. If you specify a value other than the three listed before, the value you enter (custom) will be used as the CRLF character(s)
generateHeaderColumns	boolean	optional - default value = false - uses to generate the header columns of the CSV generates
autospanLine	boolean	Camel 2.13/2.12.2: optional - default value = false - if enabled then the last column auto spanned to end of line, for example if its a comment, etc this allows the line to contain all characters, also the delimiter char.
isOrdered	boolean	optional - default value = false - allow to change the order of the fields when CSV is generated
quote	String	Camel 2.8.3/2.9: option - allow to specify a quote character of the fields when CSV is generated
quoting	boolean	This annotation is associated to the root class of the model and must be declared one time. Camel 2.11: optional - default value = false - Indicate if the values must be quoted when marshaling when CSV is generated.

case 1 : separator = ','

The separator used to segregate the fields in the CSV record is ',' :

10, J, Pauline, M, XD12345678, Fortis Dynamic 15/15, 2500, USD,08-01-2009

```
@CsvRecord( separator = "," )
public class Order {
...
}
```

case 2 : separator = ';'

Compare to the previous case, the separator here is ';' instead of ',' :

10; J; Pauline; M; XD12345678; Fortis Dynamic 15/15; 2500; USD; 08-01-2009

```
@CsvRecord( separator = ";" )
public class Order {
...
}
```

case 3 : separator = '|'

Compare to the previous case, the separator here is '|' instead of ',' :

10| J| Pauline| M| XD12345678| Fortis Dynamic 15/15| 2500| USD| 08-01-2009

```
@CsvRecord( separator = "\\|\" )
public class Order {
...
}
```

case 4 : separator = '\"'\"'**Applies for Camel 2.8.2 or older**

When the field to be parsed of the CSV record contains ',' or ';' which is also used as separator, we should find another strategy

to tell camel bindy how to handle this case. To define the field containing the data with a comma, you will use simple or double quotes

as delimiter (e.g : '10', 'Street 10, NY', 'USA' or "10", "Street 10, NY", "USA").

Remark : In this case, the first and last character of the line which are a simple or double quotes will be removed by bindy

"10","J","Pauline","M","XD12345678","Fortis Dynamic 15,15" 2500,"USD","08-01-2009"

```
@CsvRecord( separator = "\",\"" )
public class Order {
...
}
```

From **Camel 2.8.3/2.9 or never** bindy will automatically detect if the record is enclosed with either single or double quotes and automatically remove those quotes when unmarshalling from CSV to Object. Therefore do not include the quotes in the separator, but simply do as below:

"10","J","Pauline","M","XD12345678","Fortis Dynamic 15,15" 2500,"USD","08-01-2009"

```
@CsvRecord( separator = "," )
public class Order {
...
}
```

Notice that if you want to marshal from Object to CSV and use quotes, then you need to specify which quote character to use, using the quote attribute on the @CsvRecord as shown below:

```
@CsvRecord( separator = ",", quote = "\"\"\" )
public class Order {
...
}
```

case 5 : separator & skipfirstline

The feature is interesting when the client wants to have in the first line of the file, the name of the data fields :

order id, client id, first name, last name, isin code, instrument name, quantity, currency, date

To inform bindy that this first line must be skipped during the parsing process, then we use the attribute :

```
@CsvRecord(separator = ",", skipFirstLine = true)
public class Order {
...
}
```

case 6 : generateHeaderColumns

To add at the first line of the CSV generated, the attribute generateHeaderColumns must be set to true in the annotation like this :

```
@CsvRecord( generateHeaderColumns = true )
```

```
public Class Order {
    ...
}
```

As a result, Bindy during the unmarshaling process will generate CSV like this :

```
order id, client id, first name, last name, isin code, instrument name, quantity, currency, date
10, J, Pauline, M, XD12345678, Fortis Dynamic 15/15, 2500, USD,08-01-2009
```

case 7 : carriage return

If the platform where camel-bindy will run is not Windows but Macintosh or Unix, than you can change the crlf property like this. Three values are available : WINDOWS, UNIX or MAC

```
@CsvRecord(separator = ",", crlf="MAC")
public Class Order {
    ...
}
```

Additionally, if for some reason you need to add a different line ending character, you can opt to specify it using the crlf parameter. In the following example, we can end the line with a comma followed by the newline character:

```
@CsvRecord(separator = ",", crlf=",\n")
public Class Order {
    ...
}
```

case 8 : isOrdered

Sometimes, the order to follow during the creation of the CSV record from the model is different from the order used during the parsing. Then, in this case, we can use the attribute isOrdered = true to indicate this in combination with attribute 'position' of the DataField annotation.

```
@CsvRecord(isOrdered = true)
public Class Order {
    @DataField(pos = 1, position = 11)
    private int orderNr;

    @DataField(pos = 2, position = 10)
    private String clientNr;

    ...
}
```

Remark : pos is used to parse the file, stream while positions is used to generate the CSV

2. Link

The link annotation will allow to link objects together.

Annotation name	Record type	Level
Link	all	Class & Property
Parameter name	type	Info
linkType	LinkType	optional - by default the value is LinkType.oneToOne - so you are not obliged to mention it Only one-to-one relation is allowed.

e.g : If the model Class Client is linked to the Order class, then use annotation Link in the Order class like this :

```
Listing 1. Property Link

@CsvRecord(separator = ",")
public class Order {
    ...
    @DataField(pos = 1)
    private int orderNr;

    @Link
    private Client client;
    ...
}
```

AND for the class Client :

```
Listing 1. Class Link

@Link
public class Client {
    ...
}
```

3. DataField

The DataField annotation defines the property of the field. Each datafield is identified by its position in the record, a type (string, int, date, ...) and optionally of a pattern

Annotation name	Record type	Level
DataField	all	Property
Parameter name	type	Info
pos	int	mandatory - The input position of the field. digit number starting from 1 to ... - See the position parameter.
pattern	string	optional - default value = "" - will be used to format Decimal, Date, ...
length	int	optional - represents the length of the field for fixed length format
precision	int	optional - represents the precision to be used when the Decimal number will be formatted/parsed
pattern	string	optional - default value = "" - is used by the Java formatter (SimpleDateFormat by example) to format/validate data. If using pattern, then setting locale on bindy data format is recommended. Either set to a known locale such as "us" or use "default" to use platform default locale. Notice that "default" requires Camel 2.14/2.13.3/2.12.5.
position	int	optional - must be used when the position of the field in the CSV generated (output message) must be different compare to input position (pos). See the pos parameter.
required	boolean	optional - default value = "false"
trim	boolean	optional - default value = "false"
defaultValue	string	Camel 2.10: optional - default value = "" - defines the field's default value when the respective CSV field is empty/not available
impliedDecimalSeparator	boolean	Camel 2.11: optional - default value = "false" - Indicates if there is a decimal point implied at a specified location
lengthPos	int	Camel 2.11: optional - can be used to identify a data field in a fixed-length record that defines the fixed length for this field
delimiter	string	Camel 2.11: optional - can be used to demarcate the end of a variable-length field within a fixed-length record

case 1 : pos

This parameter/attribute represents the position of the field in the csv record

```
Listing 1. Position
@CsvRecord(separator = ",")
public class Order {
    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 5)
    private String isinCode;
}
```

As you can see in this example the position starts at '1' but continues at '5' in the class Order. The numbers from '2' to '4' are defined in the class Client (see here after).

```
Listing 1. Position continues in another model class
public class Client {
    @DataField(pos = 2)
    private String clientNr;

    @DataField(pos = 3)
    private String firstName;

    @DataField(pos = 4)
    private String lastName;
}
```

case 2 : pattern

The pattern allows to enrich or validates the format of your data

```
Listing 1. Pattern
@CsvRecord(separator = ",")
public class Order {
    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 5)
    private String isinCode;

    @DataField(name = "Name", pos = 6)
    private String instrumentName;

    @DataField(pos = 7, precision = 2)
    private BigDecimal amount;
```

```

@DataField(pos = 8)
private String currency;

@DataField(pos = 9, pattern = "dd-MM-yyyy") -- pattern used during parsing or when the date is created
private Date orderDate;
...
}

```

case 3 : precision

The precision is helpful when you want to define the decimal part of your number

Listing 1. Precision

```

@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @Link
    private Client client;

    @DataField(pos = 5)
    private String isinCode;

    @DataField(name = "Name", pos = 6)
    private String instrumentName;

    @DataField(pos = 7, precision = 2) -- precision
    private BigDecimal amount;

    @DataField(pos = 8)
    private String currency;

    @DataField(pos = 9, pattern = "dd-MM-yyyy")
    private Date orderDate;
...
}

```

case 4 : Position is different in output

The position attribute will inform bindy how to place the field in the CSV record generated. By default, the position used corresponds to the position defined with the attribute 'pos'. If the position is different (that means that we have an asymmetric processus comparing marshaling from unmarshaling) than we can use 'position' to indicate this.

Here is an example

```

Listing 1. Position is different in output

@CsvRecord(separator = ",")
public class Order {
    @CsvRecord(separator = ",", isOrdered = true)
    public class Order {

        // Positions of the fields start from 1 and not from 0

        @DataField(pos = 1, position = 11)
        private int orderNr;

        @DataField(pos = 2, position = 10)
        private String clientNr;

        @DataField(pos = 3, position = 9)
        private String firstName;

        @DataField(pos = 4, position = 8)
        private String lastName;

        @DataField(pos = 5, position = 7)
        private String instrumentCode;

        @DataField(pos = 6, position = 6)
        private String instrumentNumber;
    }
}

```

This attribute of the annotation `@DataField` must be used in combination with attribute `isOrdered = true` of the annotation `@CsvRecord`

case 5 : required

If a field is mandatory, simply use the attribute 'required' setted to true

Listing 1. Required

```

@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 2, required = true)
    private String clientNr;

    @DataField(pos = 3, required = true)
    private String firstName;

    @DataField(pos = 4, required = true)
    private String lastName;
...
}

```

If this field is not present in the record, than an error will be raised by the parser with the following information :

Some fields are missing (optional or mandatory), line :

case 6 : trim

If a field has leading and/or trailing spaces which should be removed before they are processed, simply use the attribute 'trim' setted to true

Listing 1. Trim

```
@CsvRecord(separator = ",")  
public class Order {  
  
    @DataField(pos = 1, trim = true)  
    private int orderNr;  
  
    @DataField(pos = 2, trim = true)  
    private Integer clientNr;  
  
    @DataField(pos = 3, required = true)  
    private String firstName;  
  
    @DataField(pos = 4)  
    private String lastName;  
}
```

case 7 : defaultValue

If a field is not defined then uses the value indicated by the defaultValue attribute

Listing 1. Default value

```
@CsvRecord(separator = ",")  
public class Order {  
  
    @DataField(pos = 1)  
    private int orderNr;  
  
    @DataField(pos = 2)  
    private Integer clientNr;  
  
    @DataField(pos = 3, required = true)  
    private String firstName;  
  
    @DataField(pos = 4, defaultValue = "Barin")  
    private String lastName;  
}
```

This attribute is only applicable to optional fields.

4. FixedLengthRecord

The FixedLengthRecord annotation is used to identified the root class of the model. It represents a record = a line of a file/message containing data fixed length formatted and can be linked to several children model classes. This format is a bit particular because data of a field can be aligned to the right or to the left.

When the size of the data does not fill completely the length of the field, we can then add 'padd' characters.

Annotation name	Record type	Level
FixedLengthRecord	fixed	Class
Parameter name	type	Info
crlf	string	optional - possible values = WINDOWS,UNIX,MAC, or custom; default value = WINDOWS - allow to define the carriage return character to use. If you specify a value other than the three listed before, the value you enter (custom) will be used as the CRLF character(s)
paddingChar	char	mandatory - default value = ''
length	int	mandatory = size of the fixed length record
hasHeader	boolean	Camel 2.11 - optional - Indicates that the record(s) of this type may be preceded by a single header record at the beginning of the file / stream
hasFooter	boolean	Camel 2.11 - optional - Indicates that the record(s) of this type may be followed by a single footer record at the end of the file / stream
skipHeader	boolean	Camel 2.11 - optional - Configures the data format to skip marshalling / unmarshalling of the header record. Configure this parameter on the primary record (e.g., not the header or footer).
skipFooter	boolean	Camel 2.11 - optional - Configures the data format to skip marshalling / unmarshalling of the footer record. Configure this parameter on the primary record (e.g., not the header or footer)..
isHeader	boolean	Camel 2.11 - optional - Identifies this FixedLengthRecord as a header record
isFooter	boolean	Camel 2.11 - optional - Identifies this FixedLengthRecords as a footer record
ignoreTrailingChars	boolean	Camel 2.11.1 - optional - Indicates that characters beyond the last mapped filed can be

ignored when unmarshalling / parsing.

This annotation is associated to the root class of the model and must be declared one time.

The hasHeader/hasFooter parameters are mutually exclusive with isHeader/isFooter. A record may not be both a header/footer and a primary fixed-length record.

case 1 : Simple fixed length record

This simple example shows how to design the model to parse/format a fixed message

10A9PaulineMISINXD12345678BUYShare2500.45USD01-08-2009

Listing 1. Fixed-simple

```
@FixedLengthRecord(length=54, paddingChar=' ');
public static class Order {
    @DataField(pos = 1, length=2)
    private int orderNr;

    @DataField(pos = 3, length=2)
    private String clientNr;

    @DataField(pos = 5, length=7)
    private String firstName;

    @DataField(pos = 12, length=1, align="L")
    private String lastName;

    @DataField(pos = 13, length=4)
    private String instrumentCode;

    @DataField(pos = 17, length=10)
    private String instrumentNumber;

    @DataField(pos = 27, length=3)
    private String orderType;

    @DataField(pos = 30, length=5)
    private String instrumentType;

    @DataField(pos = 35, precision = 2, length=7)
    private BigDecimal amount;

    @DataField(pos = 42, length=3)
    private String currency;

    @DataField(pos = 45, length=10, pattern = "dd-MM-yyyy")
    private Date orderDate;
    ...
}
```

case 2 : Fixed length record with alignment and padding

This more elaborated example show how to define the alignment for a field and how to assign a padding character which is '' here''

10A9 PaulineM ISINXD12345678BUYShare2500.45USD01-08-2009

Listing 1. Fixed-padding-align

```
@FixedLengthRecord(length=60, paddingChar=' ', pattern="^");
public static class Order {
    @DataField(pos = 1, length=2)
    private int orderNr;

    @DataField(pos = 3, length=2)
    private String clientNr;

    @DataField(pos = 5, length=9)
    private String firstName;

    @DataField(pos = 14, length=5, align="L") // align text to the LEFT zone of the block
    private String lastName;

    @DataField(pos = 19, length=4)
    private String instrumentCode;

    @DataField(pos = 23, length=10)
    private String instrumentNumber;

    @DataField(pos = 33, length=3)
    private String orderType;

    @DataField(pos = 36, length=5)
    private String instrumentType;

    @DataField(pos = 41, precision = 2, length=7)
    private BigDecimal amount;

    @DataField(pos = 48, length=3)
    private String currency;

    @DataField(pos = 51, length=10, pattern = "dd-MM-yyyy")
    private Date orderDate;
    ...
}
```

case 3 : Field padding

Sometimes, the default padding defined for record cannot be applied to the field as we have a number format where we would like to pad with '0' instead of''. In this case, you can use in the model the attribute paddingField to set this value.

10A9 PaulineM ISINXD12345678BUYShare000002500.45USD01-08-2009

Listing 1. Fixed-padding-field

```
@FixedLengthRecord(length = 65, paddingChar = ' ')
public static class Order {
    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 3, length = 2)
    private String clientNr;

    @DataField(pos = 5, length = 9)
    private String firstName;

    @DataField(pos = 14, length = 5, align = "L")
    private String lastName;

    @DataField(pos = 19, length = 4)
    private String instrumentCode;

    @DataField(pos = 23, length = 10)
    private String instrumentNumber;

    @DataField(pos = 33, length = 3)
    private String orderType;

    @DataField(pos = 36, length = 5)
    private String instrumentType;

    @DataField(pos = 41, precision = 2, length = 12, paddingChar = '0')
    private BigDecimal amount;

    @DataField(pos = 53, length = 3)
    private String currency;

    @DataField(pos = 56, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
    ...
}
```

case 4: Fixed length record with delimiter

Fixed-length records sometimes have delimited content within the record. The firstName and lastName fields are delimited with the '^' character in the following example:

10A9Pauline^M^ISINXD12345678BUYShare000002500.45USD01-08-2009

Listing 1. Fixed-delimited

```
@FixedLengthRecord()
public static class Order {
    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 2, length = 2)
    private String clientNr;

    @DataField(pos = 3, delimiter = "^")
    private String firstName;

    @DataField(pos = 4, delimiter = "^")
    private String lastName;

    @DataField(pos = 5, length = 4)
    private String instrumentCode;

    @DataField(pos = 6, length = 10)
    private String instrumentNumber;

    @DataField(pos = 7, length = 3)
    private String orderType;

    @DataField(pos = 8, length = 5)
    private String instrumentType;

    @DataField(pos = 9, precision = 2, length = 12, paddingChar = '0')
    private BigDecimal amount;

    @DataField(pos = 10, length = 3)
    private String currency;

    @DataField(pos = 11, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
}
```

As of Camel 2.11 the 'pos' value(s) in a fixed-length record may optionally be defined using ordinal, sequential values instead of precise column numbers.

case 5 : Fixed length record with record-defined field length

Occasionally a fixed-length record may contain a field that define the expected length of another field within the same record. In the following example the length of the instrumentNumber field value is defined by the value of instrumentNumberLen field in the record.

10A9Pauline^M^ISIN10XD12345678BUYShare000002500.45USD01-08-2009

Listing 1. Fixed-delimited

```
@FixedLengthRecord()
public static class Order {
    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 2, length = 2)
    private String clientNr;
```

```

@DataField(pos = 3, delimiter = "^")
private String firstName;

@DataField(pos = 4, delimiter = "^")
private String lastName;

@DataField(pos = 5, length = 4)
private String instrumentCode;

@DataField(pos = 6, length = 2, align = "R", paddingChar = '#;0#')
private int instrumentNumberLen;

@DataField(pos = 7, lengthPos=6)
private String instrumentNumber;

@DataField(pos = 8, length = 3)
private String orderType;

@DataField(pos = 9, length = 5)
private String instrumentType;

@DataField(pos = 10, precision = 2, length = 12, paddingChar = '#;0#')
private BigDecimal amount;

@DataField(pos = 11, length = 3)
private String currency;

@DataField(pos = 12, length = 10, pattern = "dd-MM-yyyy")
private Date orderDate;

```

case 6 : Fixed length record with header and footer

Bindy will discover fixed-length header and footer records that are configured as part of the model – provided that the annotated classes exist either in the same package as the primary `@FixedLengthRecord` class, or within one of the configured scan packages. The following text illustrates two fixed-length records that are bracketed by a header record and footer record.

```

101-08-2009
10A9 PaulineM ISINXD12345678BUYShare000002500.45USD01-08-2009
10A9 RichN ISINXD12345678BUYShare000002700.45USD01-08-2009
9000000002

```

```

Listing 1. Fixed-header-and-footer-main-class

@fixedLengthRecord(hasHeader = true, hasFooter = true)
public class Order {

    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 2, length = 2)
    private String clientNr;

    @DataField(pos = 3, length = 9)
    private String firstName;

    @DataField(pos = 4, length = 5, align = "L")
    private String lastName;

    @DataField(pos = 5, length = 4)
    private String instrumentCode;

    @DataField(pos = 6, length = 10)
    private String instrumentNumber;

    @DataField(pos = 7, length = 3)
    private String orderType;

    @DataField(pos = 8, length = 5)
    private String instrumentType;

    @DataField(pos = 9, precision = 2, length = 12, paddingChar = '#;0#')
    private BigDecimal amount;

    @DataField(pos = 10, length = 3)
    private String currency;

    @DataField(pos = 11, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
}

@FixedLengthRecord(isHeader = true)
public class OrderHeader {
    @DataField(pos = 1, length = 1)
    private int recordType;
}

@DataField(pos = 2, length = 10, pattern = "dd-MM-yyyy")
private Date recordDate;

...

@FixedLengthRecord(isFooter = true)
public class OrderFooter {
    @DataField(pos = 1, length = 1)
    private int recordType;

    @DataField(pos = 2, length = 9, align = "R", paddingChar = '#;0#')
    private int numberOfRecordsInThefile;
}

...

```

case 7 : Skipping content when parsing a fixed length record. (Camel 2.11.1)

It is common to integrate with systems that provide fixed-length records containing more information than needed for the target use case. It is useful in this situation to skip the declaration and parsing of those fields that we do not need. To accommodate this, Bindy will skip forward to the next mapped field within a record if the 'pos' value of the next declared field is beyond the cursor position of the last parsed field. Using absolute 'pos' locations for the fields of interest (instead of ordinal values) causes Bindy to skip content between two fields.

Similarly, it is possible that none of the content beyond some field is of interest. In this case, you can tell Bindy to skip parsing of everything beyond the last mapped field by setting the **ignoreTrailingChars** property on the **@FixedLengthRecord** declaration.

```
@FixedLengthRecord(ignoreTrailingChars = true)
public static class Order {
    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 3, length = 2)
    private String clientNr;

    ... any characters that appear beyond the last mapped field will be ignored
}
```

5. Message

The Message annotation is used to identify the class of your model who will contain key value pairs fields. This kind of format is used mainly in Financial Exchange Protocol Messages (FIX). Nevertheless, this annotation can be used for any other format where data are identified by keys. The key pair values are separated each other by a separator which can be a special character like a tab delimiter (unicode representation : \u0009) or a start of heading (unicode representation : \u0001) "FIX information"

More information about FIX can be found on this web site : <http://www.fixprotocol.org/>. To work with FIX messages, the model must contain a Header and Trailer classes linked to the root message class which could be a Order class. This is not mandatory but will be very helpful when you will use camel-bindy in combination with camel-fix which is a Fix gateway based on quickFix project <http://www.quickfixj.org/>.

Annotation name	Record type	Level
Message	key value pair	Class
pairSeparator	string	mandatory - can be '=' or ';' or 'anything'
keyValuePairSeparat	string	mandatory - can be '\u0001', '\u0009', '#' or 'anything'
crlf	string	optional - possible values = WINDOWS,UNIX,MAC, or custom; default value = WINDOWS - allow to define the carriage return character to use. If you specify a value other than the three listed before, the value you enter (custom) will be used as the CRLF character(s)
type	string	optional - define the type of message (e.g. FIX, EMX, ...)
version	string	optional - version of the message (e.g. 4.1)
isOrdered	boolean	optional - default value = false - allow to change the order of the fields when FIX message is generated
This annotation is associated to the message class of the model and must be declared one time.		

case 1 : separator = '\u0001'

The separator used to segregate the key value pair fields in a FIX message is the ASCII '01' character or in unicode format '\u0001'. This character must be escaped a second time to avoid a java runtime error. Here is an example :

8=FIX.4.1 9=20 34=1 35=0 49=INVMGR 56=BRKR 1=BE.CHM.001 11=CHM0001-01 22=4 ...

and how to use the annotation

```
Listing 1. FIX - message

@Message(keyValuePairSeparator = "=", pairSeparator = "\u0001", type="FIX", version="4.1")
public class Order {
    ...
}
```

Look at test cases

The ASCII character like tab, ... cannot be displayed in WIKI page. So, have a look to the test case of camel-bindy to see exactly how the FIX message looks like (src\test\data\fix\fix.txt) and the Order, Trailer, Header classes (src\test\java\org\apache\camel\dataformat\bindy\model\fix\simple\Order.java)

6. KeyValuePairField

The `KeyValuePairField` annotation defines the property of a key value pair field. Each `KeyValuePairField` is identified by a tag (= key) and its value associated, a type (string, int, date, ...), optionally a pattern and if the field is required

Annotation name	Record type	Level
<code>KeyValuePairField</code>	Key Value Pair - FIX	Property
Parameter name	type	Info
tag	int	mandatory - digit number identifying the field in the message - must be unique
pattern	string	optional - default value = "" - will be used to format Decimal, Date, ...
precision	int	optional - digit number - represents the precision to be used when the Decimal number will be formatted/parsed
position	int	optional - must be used when the position of the key/tag in the FIX message must be different
required	boolean	optional - default value = "false"
impliedDecimalSeparator	boolean	Camel 2.11: optional - default value = "false" - Indicates if there is a decimal point implied at a specified location

case 1 : tag

This parameter represents the key of the field in the message

```
Listing 1. FIX message - Tag

@Message(keyValuePairSeparator = "=", pairSeparator = "\u0001", type="FIX", version="4.1")
public class Order {

    @Link Header header;
    @Link Trailer trailer;

    @KeyValuePairField(tag = 1) // Client reference
    private String Account;

    @KeyValuePairField(tag = 11) // Order reference
    private String c1OrId;

    @KeyValuePairField(tag = 22) // Fund ID type (Sedol, ISIN, ...)
    private String IDSource;

    @KeyValuePairField(tag = 48) // Fund code
    private String SecurityId;

    @KeyValuePairField(tag = 54) // Movement type ( 1 = Buy, 2 = sell)
    private String Side;

    @KeyValuePairField(tag = 58) // Free text
    private String Text;

    ...
}
```

case 2 : Different position in output

If the tags/keys that we will put in the FIX message must be sorted according to a predefined order, then use the attribute 'position' of the annotation `@KeyValuePairField`

```
Listing 1. FIX message - Tag - sort

@Message(keyValuePairSeparator = "=", pairSeparator = "\u0001", type = "FIX", version = "4.1", isOrdered = true)
public class Order {

    @Link Header header;
    @Link Trailer trailer;

    @KeyValuePairField(tag = 1, position = 1) // Client reference
    private String account;

    @KeyValuePairField(tag = 11, position = 3) // Order reference
    private String c1OrId;

    ...
}
```

7. Section

In FIX message of fixed length records, it is common to have different sections in the representation of the information : header, body and section. The purpose of the annotation `@Section` is to inform bindy about which class of the model represents the header (= section 1), body (= section 2) and footer (= section 3)

Only one attribute/parameter exists for this annotation.

Annotation name	Record type	Level
<code>Section</code>	FIX	Class
Parameter name	type	Info

number	int	digit number identifying the section position
--------	-----	---

case 1 : Section

A. Definition of the header section

```
Listing 1. FIX message - Section - Header

@Section(number = 1)
public class Header {

    @KeyValuePairField(tag = 8, position = 1) // Message Header
    private String beginString;

    @KeyValuePairField(tag = 9, position = 2) // Checksum
    private int bodyLength;
}
```

B. Definition of the body section

```
Listing 1. FIX message - Section - Body

@Section(number = 2)
@Message(keyValuePairSeparator = "=", pairSeparator = "\u0001", type = "FIX", version = "4.1", isOrdered = true)
public class Order {

    @Link Header header;

    @Link Trailer trailer;

    @KeyValuePairField(tag = 1, position = 1) // Client reference
    private String account;

    @KeyValuePairField(tag = 11, position = 3) // Order reference
    private String clOrdId;
}
```

C. Definition of the footer section

```
Listing 1. FIX message - Section - Footer

@Section(number = 3)
public class Trailer {

    @KeyValuePairField(tag = 10, position = 1)
    // Checksum
    private int checkSum;

    public int getCheckSum() {
        return checkSum;
    }
}
```

8. OneToMany

The purpose of the annotation `@OneToMany` is to allow to work with a `List<?>` field defined a POJO class or from a record containing repetitive groups.

Restrictions `OneToMany`

Be careful, the one to many of bindy does not allow to handle repetitions defined on several levels of the hierarchy

The relation `OneToMany` ONLY WORKS in the following cases :

- Reading a FIX message containing repetitive groups (= group of tags/keys)
- Generating a CSV with repetitive data

Annotation name	Record type	Level
<code>OneToMany</code>	all	property

Parameter name	type	Info
<code>mappedTo</code>	string	optional - string - class name associated to the type of the <code>List<Type of the Class></code>

case 1 : Generating CSV with repetitive data

Here is the CSV output that we want :

Claus,Ibsen,Camel in Action 1,2010,35

Claus,Ibsen,Camel in Action 2,2012,35

Claus,Ibsen,Camel in Action 3,2013,35

Claus,Ibsen,Camel in Action 4,2014,35

Remark : the repetitive data concern the title of the book and its publication date while first, last name and age are common and the classes used to modeling this. The Author class contains a List of Book.

```
Listing 1. Generate CSV with repetitive data

@CsvRecord(separator=",")
public class Author {

    @DataField(pos = 1)
    private String firstName;

    @DataField(pos = 2)
```

```

private String lastName;
@OneToMany
private List<Book> books;
@DataField(pos = 5)
private String Age;
...

public class Book {
    @DataField(pos = 3)
    private String title;
    @DataField(pos = 4)
    private String year;
}

```

Very simple isn't it !!!

case 2 : Reading FIX message containing group of tags/keys

Here is the message that we would like to process in our model :

```

"8=FIX 4.19=2034=135=049=INVGR56=BRKR"
"1=BE.CHM.00111=CHM0001-0158=this is a camel - bindy test"
"22=448=BE000124567854=1"
"22=548=BE000987654354=2"
"22=648=BE000999999954=3"
"10=220"

```

tags 22, 48 and 54 are repeated
and the code

```

Listing 1. Reading FIX message containing group of tags/keys

public class Order {
    @Link Header header;
    @Link Trailer trailer;
    @KeyValuePairField(tag = 1) // Client reference
    private String account;
    @KeyValuePairField(tag = 11) // Order reference
    private String clOrdId;
    @KeyValuePairField(tag = 58) // Free text
    private String text;
    @OneToMany(mappedBy = "org.apache.camel.dataformat.bindy.model.fix.complex.onetomany.Security")
    List<Security> securities;
    ...

    public class Security {
        @KeyValuePairField(tag = 22) // Fund ID type (Sedol, ISIN, ...)
        private String idSource;
        @KeyValuePairField(tag = 48) // Fund code
        private String securityCode;
        @KeyValuePairField(tag = 54) // Movement type ( 1 = Buy, 2 = sell)
        private String side;
    }
}

```

Using the Java DSL

The next step consists in instantiating the DataFormat *bindy* class associated with this record type and providing Java package name(s) as parameter.

For example the following uses the class *BindyCsvDataFormat* (who correspond to the class associated with the CSV record type) which is configured with "com.acme.model" package name to initialize the model objects configured in this package.

```

// Camel 2.15 or older (configure by package name)
DataFormat bindy = new BindyCsvDataFormat("com.acme.model");

Â
// Camel 2.16 onwards (configure by class name)
DataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);

```

Setting locale

Bindy supports configuring the locale on the dataformat, such as

```

// Camel 2.15 or older (configure by package name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat("com.acme.model");
// Camel 2.16 onwards (configure by class name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);

bindy.setLocale("us");

```

Or to use the platform default locale then use "default" as the locale name. Notice this requires Camel 2.14/2.13.3/2.12.5.

```
// Camel 2.15 or older (configure by package name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat("com.acme.model");
// Camel 2.16 onwards (configure by class name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);
bindy.setLocale("default");
```

for older releases you can set it using Java code as shown

```
// Camel 2.15 or older (configure by package name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat("com.acme.model");
// Camel 2.16 onwards (configure by class name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);

bindy.setLocale(Locale.getDefault().getISO3Country());
```

Unmarshaling

```
from("file://inbox")
.unmarshal(bindy)
.to("direct:handleOrders");
```

Alternatively, you can use a named reference to a data format which can then be defined in your Registry e.g. your Spring XML file:

```
from("file://inbox")
.unmarshal("myBindyDataFormat")
.to("direct:handleOrders");
```

The Camel route will pick-up files in the inbox directory, unmarshall CSV records into a collection of model objects and send the collection to the route referenced by 'handleOrders'.

The collection returned is a **List of Map** objects. Each Map within the list contains the model objects that were marshalled out of each line of the CSV. The reason behind this is that *each line can correspond to more than one object*. This can be confusing when you simply expect one object to be returned per line.

Each object can be retrieve using its class name.

```
List<Map<String, Object>> unmarshaledModels = (List<Map<String, Object>>) exchange.getIn().getBody();

int modelCount = 0;
for (Map<String, Object> model : unmarshaledModels) {
    for (String className : model.keySet()) {
        Object obj = model.get(className);
        LOG.info("Count : " + modelCount + ", " + obj.toString());
    }
    modelCount++;
}

LOG.info("Total CSV records received by the csv bean : " + modelCount);
```

Assuming that you want to extract a single Order object from this map for processing in a route, you could use a combination of a Splitter and a Processor as per the following:

```
from("file://inbox")
.unmarshal(bindy)
.split(body())
.process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        Message in = exchange.getIn();
        Map<String, Object> modelMap = (Map<String, Object>) in.getBody();
        in.setBody(modelMap.get(Order.class.getCanonicalName()));
    }
})
.to("direct:handleSingleOrder")
.end();
```

Take care of the fact that Bindy uses CHARSET_NAME property or the CHARSET_NAME header as define in the Exchange interface to do a characterset conversion of the inputstream received for unmarshalling. In some producers (e.g. file-endpoint) you can define a characterset. The characterset conversion can already been done by this producer. Sometimes you need to remove this property or header from the exchange before sending it to the unmarshal. If you don't remove it the conversion might be done twice which might lead to unwanted results.

```
from("file://inbox?charset=Cp922")
.removeProperty(Exchange.CHARSET_NAME)
.unmarshal("myBindyDataFormat")
.to("direct:handleOrders");
```

Marshaling

To generate CSV records from a collection of model objects, you create the following route :

```
from("direct:handleOrders")
    .marshal(bindy)
    .to("file://outbox")
```

Using Spring XML

This is really easy to use Spring as your favorite DSL language to declare the routes to be used for camel-bindy. The following example shows two routes where the first will pick-up records from files, unmarshal the content and bind it to their model. The result is then send to a pojo (doing nothing special) and place them into a queue.

The second route will extract the pojos from the queue and marshal the content to generate a file containing the csv record. The example above is for using Camel 2.16 onwards.

Listing 1. spring_dsl

```
<?xml version="1.0" encoding="UTF-8"?>

<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.xsd
        http://camel.apache.org/schema/spring
        http://camel.apache.org/schema/spring/camel-spring.xsd"

    <!-- Queuing engine - ActiveMq - work locally in mode virtual memory -->
    <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
        <property name="brokerURL" value="vm://localhost:61616"/>
    </bean>

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <!--
            <dataFormats>
                <bindy id="bindyDataformat" type="Csv" classType="org.apache.camel.bindy.model.Order"/>
            </dataFormats>

            <route>
                <from uri="file://src/data/csv/?noop=true" />
                <unmarshal ref="bindyDataformat" />
                <to uri="bean:csv" />
                <to uri="activemq:queue:in" />
            </route>

            <route>
                <from uri="activemq:queue:in" />
                <marshal ref="bindyDataformat" />
                <to uri="file://src/data/csv/out/" />
            </route>
        </camelContext>
    </beans>
```

Be careful

Please verify that your model classes implements serializable otherwise the queue manager will raise an error

Dependencies

To use Bindy in your camel routes you need to add the a dependency on **camel-bindy** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-bindy</artifactId>
    <version>x.x.x</version>
</dependency>
```

XMLSECURITY DATA FORMAT

The XMLSecurity Data Format facilitates encryption and decryption of XML payloads at the Document, Element, and Element Content levels (including simultaneous multi-node encryption/decryption using XPath). To sign messages using the XML Signature specification, please see the Camel XML Security component.

The encryption capability is based on formats supported using the Apache XML Security (Santuario) project. Symmetric encryption/decryption is currently supported using Triple-DES and AES (128, 192, and 256) encryption formats. Additional formats can be easily added later as needed. This capability allows Camel users to encrypt/decrypt payloads while being dispatched or received along a route.

Available as of Camel 2.9

The XMLSecurity Data Format supports asymmetric key encryption. In this encryption model a symmetric key is generated and used to perform XML content encryption or decryption. This "content encryption key" is then itself encrypted using an asymmetric encryption algorithm that leverages the recipient's public key as the "key encryption key". Use of an asymmetric key encryption algorithm ensures that only the holder of the recipient's private key can access the generated symmetric encryption key. Thus,

only the private key holder can decode the message. The XMLSecurity Data Format handles all of the logic required to encrypt and decrypt the message content and encryption key(s) using asymmetric key encryption.

The XMLSecurity Data Format also has improved support for namespaces when processing the XPath queries that select content for encryption. A namespace definition mapping can be included as part of the data format configuration. This enables true namespace matching, even if the prefix values in the XPath query and the target xml document are not equivalent strings.

Basic Options

Option	Default	Description
secureTag	null	The XPath reference to the XML Element selected for encryption/decryption. If no tag is specified, the entire payload is encrypted/decrypted.
secureTagContents	false	A boolean value to specify whether the XML Element is to be encrypted or the contents of the XML Element <ul style="list-style-type: none"> • false = Element Level • true = Element Content Level
passPhrase	null	A String used as passPhrase to encrypt/decrypt content. The passPhrase has to be provided. If no passPhrase is specified, a default passPhrase is used. The passPhrase needs to be put together in conjunction with the appropriate encryption algorithm. For example using TRIPLEDES the passPhrase can be a "only another 24 Byte key"
xmlCipherAlgorithm	TRIPLEDES	The cipher algorithm to be used for encryption/decryption of the XML message content. The available choices are: <ul style="list-style-type: none"> • XMLCipher.TRIPLEDES • XMLCipher.AES_128 • XMLCipher.AES_128_GCM Camel 2.12 • XMLCipher.AES_192 • XMLCipher.AES_192_GCM Camel 2.12 • XMLCipher.AES_256 • XMLCipher.AES_256_GCM Camel 2.12 • XMLCipher.SEED_128 Camel 2.15 • XMLCipher.CAMELLIA_128, XMLCipher.CAMELLIA_192, XMLCipher.CAMELLIA_256 Camel 2.15
namespaces	null	A map of namespace values indexed by prefix. The index values must match the prefixes used in the secureTag XPath query.

Asymmetric Encryption Options

These options can be applied in addition to relevant the Basic options to use asymmetric key encryption.

Option	Default	Description
recipientKeyAlias	null	The key alias to be used when retrieving the recipient's public or private key from a KeyStore when performing asymmetric key encryption or decryption.
keycipherAlgorithm	Camel 2.12 XMLCipher.RSA_OAEP	The cipher algorithm to be used for encryption/decryption of the asymmetric key. The available choices are: <ul style="list-style-type: none"> • XMLCipher.RSA_v1dot5 • XMLCipher.RSA_OAEP • XMLCipher.RSA_OAEP_11
keyOrTrustStoreParameters	null	Configuration options for creating and loading a KeyStore instance that represents the sender's trustStore or recipient's keyStore.
keyPassword	null	Camel 2.10.2 / 2.11: The password to be used for retrieving the private key from the KeyStore. This key is used for asymmetric decryption.
digestAlgorithm	XMLCipher.SHA1	Camel 2.12 The digest algorithm to use with the RSA OAEP algorithm. The available choices are: <ul style="list-style-type: none"> • XMLCipher.SHA1 • XMLCipher.SHA256 • XMLCipher.SHA512

mgfAlgorithm	EncryptionConstants.MGF1_SHA1	Camel 2.12 The MGF Algorithm to use with the RSA OAEP algorithm. The available choices are:
addKeyValueForEncryptedKey	true	Camel 2.14.1 Whether to add the public key used to encrypt the session key as a KeyValue in the EncryptedKey structure or not.

Key Cipher Algorithm

As of Camel 2.12.0, the default Key Cipher Algorithm is now XMLCipher.RSA_OAEP instead of XMLCipher.RSA_v1dot5. Usage of XMLCipher.RSA_v1dot5 is discouraged due to various attacks. Requests that use RSA v1.5 as the key cipher algorithm will be rejected unless it has been explicitly configured as the key cipher algorithm.

Marshal

In order to encrypt the payload, the `marshal` processor needs to be applied on the route followed by the `secureXML()` tag.

Unmarshal

In order to decrypt the payload, the `unmarshal` processor needs to be applied on the route followed by the `secureXML()` tag.

Examples

Given below are several examples of how marshalling could be performed at the Document, Element, and Content levels.

Full Payload encryption/decryption

```
from("direct:start")
    .marshal().secureXML()
    .unmarshal().secureXML()
    .to("direct:end");
```

Partial Payload Content Only encryption/decryption

```
String tagXPATH = "//cheesesites/italy/cheese";
boolean secureTagContent = true;

from("direct:start")
    .marshal().secureXML(tagXPATH, secureTagContent)
    .unmarshal().secureXML(tagXPATH, secureTagContent)
    .to("direct:end");
```

Partial Multi Node Payload Content Only encryption/decryption

```
String tagXPATH = "//cheesesites/*/cheese";
boolean secureTagContent = true;

from("direct:start")
    .marshal().secureXML(tagXPATH, secureTagContent)
    .unmarshal().secureXML(tagXPATH, secureTagContent)
    .to("direct:end");
```

Partial Payload Content Only encryption/decryption with choice of passPhrase(password)

```
String tagXPATH = "//cheesesites/italy/cheese";
boolean secureTagContent = true;
...
String passPhrase = "Just another 24 Byte key";
from("direct:start")
    .marshal().secureXML(tagXPATH, secureTagContent, passPhrase)
    .unmarshal().secureXML(tagXPATH, secureTagContent, passPhrase)
    .to("direct:end");
```

Partial Payload Content Only encryption/decryption with passPhrase(password) and Algorithm

```
import org.apache.xml.security.encryption.XMLCipher;
....
```

```

String tagXPATH = "//cheesesites/italy/cheese";
boolean secureTagContent = true;
String passPhrase = "Just another 24 Byte key";
String algorithm= XMLCipher.TRIPLEDES;
from("direct:start")
    .marshal().secureXML(tagXPATH, secureTagContent, passPhrase, algorithm)
    .unmarshal().secureXML(tagXPATH, secureTagContent, passPhrase, algorithm)
    .to("direct:end");
}

```

Partial Payload Content with Namespace support

Java DSL

```

final Map<String, String> namespaces = new HashMap<String, String>();
namespaces.put("cust", "http://cheese.xmlsecurity.camel.apache.org/");
final KeyStoreParameters tsParameters = new KeyStoreParameters();
tsParameters.setPassword("password");
tsParameters.setResource("sender.ts");

context.addRoutes(new RouteBuilder() {
    public void configure() {
        from("direct:start")
            .marshal().secureXML("//cust:cheesesites/italy", namespaces, true, "recipient",
                testCipherAlgorithm, XMLCipher.RSA_v1dot5, tsParameters)
            .to("mock:encrypted");
    }
})

```

Spring XML

A namespace prefix that is defined as part of the camelContext definition can be re-used in context within the data format secureTag attribute of the secureXML element.

```

<camelContext id="springXmlSecurityDataFormatTestCamelContext"
    xmlns="http://camel.apache.org/schema/spring"
    xmlns:cheese="http://cheese.xmlsecurity.camel.apache.org/">
    <route>
        <from uri="direct://start"/>
        <marshal>
            <secureXML secureTag="//cheese:cheesesites/italy"
                secureTagContent="true"/>
        </marshal>
        ...
    </route>

```

Asymmetric Key Encryption

Spring XML Sender

```

<!-- trust store configuration -->
<camel:keyStoreParameters id="trustStoreParams" resource=".sender.ts" password="password"/>

<camelContext id="springXmlSecurityDataFormatTestCamelContext"
    xmlns="http://camel.apache.org/schema/spring"
    xmlns:cheese="http://cheese.xmlsecurity.camel.apache.org/">
    <route>
        <from uri="direct://start"/>
        <marshal>
            <secureXML secureTag="//cheese:cheesesites/italy"
                secureTagContent="true"
                xmlCipherAlgorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"
                keyCipherAlgorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"
                recipientKeyAlias="recipient"
                keyOrTrustStoreParametersId="trustStoreParams"/>
        </marshal>
        ...
    </route>

```

Spring XML Recipient

```

<!-- key store configuration -->
<camel:keyStoreParameters id="keyStoreParams" resource=".recipient.ks" password="password" />

<camelContext id="springXmlSecurityDataFormatTestCamelContext"
    xmlns="http://camel.apache.org/schema/spring"
    xmlns:cheese="http://cheese.xmlsecurity.camel.apache.org/">
    <route>
        <from uri="direct://encrypted"/>
        <unmarshal>
            <secureXML secureTag="//cheese:cheesesites/italy"
                secureTagContent="true"
                xmlCipherAlgorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"
                keyCipherAlgorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"
                recipientKeyAlias="recipient"
                keyOrTrustStoreParametersId="keyStoreParams"
                keyPassword="privateKeyPassword" />
        </unmarshal>
        ...
    </route>

```

Dependencies

This data format is provided within the **camel-xmlsecurity** component.

The GZip Data Format is a message compression and de-compression format. It uses the same deflate algorithm that is used in Zip DataFormat, although some additional headers are provided. This format is produced by popular gzip/gunzip tool. Messages marshalled using GZip compression can be unmarshalled using GZip decompression just prior to being consumed at the endpoint. The compression capability is quite useful when you deal with large XML and Text based payloads or when you read messages previously compressed using gzip tool.

Options

There are no options provided for this data format.

Marshal

In this example we marshal a regular text/XML payload to a compressed payload employing gzip compression format and send it an ActiveMQ queue called MY_QUEUE.

```
from("direct:start").marshal().gzip().to("activemq:queue:MY_QUEUE");
```

Unmarshal

In this example we unmarshal a gzipped payload from an ActiveMQ queue called MY_QUEUE to its original format, and forward it for processing to the unGzippedMessageProcessor.

```
from("activemq:queue:MY_QUEUE").unmarshal().gzip().process(new UnGzippedMessageProcessor());
```

Dependencies

This data format is provided in **camel-core** so no additional dependencies is needed.

CASTOR

Available as of Camel 2.1

Castor is a Data Format which uses the Castor XML library to unmarshal an XML payload into Java objects or to marshal Java objects into an XML payload.

As usually you can use either Java DSL or Spring XML to work with Castor Data Format.

Using the Java DSL

```
from("direct:order").
    marshal().castor().
    to("activemq:queue:order");
```

For example the following uses a named DataFormat of Castor which uses default Castor data binding features.

```
CastorDataFormat castor = new CastorDataFormat ();
from("activemq:My.Queue").
    unmarshal(castor).
    to("mqseries:Another.Queue");
```

If you prefer to use a named reference to a data format which can then be defined in your Registry such as via your Spring XML file. e.g.

```
from("activemq:My.Queue").
    unmarshal("myCastorType").
    to("mqseries:Another.Queue");
```

If you want to override default mapping schema by providing a mapping file you can set it as follows.

```
CastorDataFormat castor = new CastorDataFormat ();
castor.setMappingFile("mapping.xml");
```

Also if you want to have more control on Castor Marshaller and Unmarshaller you can access them as below.

```
castor.getMarshaller();
castor.getUnmarshaller();
```

Using Spring XML

The following example shows how to use Castor to unmarshal using Spring configuring the castor data type

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <unmarshal>
      <castor validation="true" />
    </unmarshal>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

This example shows how to configure the data type just once and reuse it on multiple routes. You have to set the <castor> element directly in <camelContext>.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <dataFormats>
    <castor id="myCastor"/>
  </dataFormats>

  <route>
    <from uri="direct:start"/>
    <marshal ref="myCastor"/>
    <to uri="direct:marshalled"/>
  </route>
  <route>
    <from uri="direct:marshalled"/>
    <unmarshal ref="myCastor"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

Options

Castor supports the following options

Option	Type	Default	Description
encoding	String	UTF-8	Encoding to use when marshalling an Object to XML
validation	Boolean	false	Whether validation is turned on or off.
mappingFile	String	null	Path to a Castor mapping file to load from the classpath.
packages	String[]	null	Add additional packages to Castor XmlContext
classNames	String[]	null	Add additional class names to Castor XmlContext

Dependencies

To use Castor in your camel routes you need to add the a dependency on **camel-castor** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-castor</artifactId>
  <version>x.x.x</version>
</dependency>
```

"Protocol Buffers - Google's data interchange format"

Available from Camel 2.2

Camel provides a Data Format to serialise between Java and the Protocol Buffer protocol. The project's site details why you may wish to choose this format over xml. Protocol Buffer is language-neutral and platform-neutral, so messages produced by your Camel routes may be consumed by other language implementations.

[API Site](#)
[Protobuf Implementation](#)
[Protobuf Java Tutorial](#)

PROTOBUF OVERVIEW

This quick overview of how to use Protobuf. For more detail see the complete tutorial

Defining the proto format

The first step is to define the format for the body of your exchange. This is defined in a .proto file as so:

```
Listing 1. addressbook.proto

package org.apache.camel.component.protobuf;

option java_package = "org.apache.camel.component.protobuf";
option java_outer_classname = "AddressBookProtos";

message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;

    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }

    message PhoneNumber {
        required string number = 1;
        optional PhoneType type = 2 [default = HOME];
    }

    repeated PhoneNumber phone = 4;
}

message AddressBook {
    repeated Person person = 1;
}
```

Generating Java classes

The Protobuf SDK provides a compiler which will generate the Java classes for the format we defined in our .proto file. You can run the compiler for any additional supported languages you require.

```
protoc --java_out=. ./addressbook.proto
```

This will generate a single Java class named AddressBookProtos which contains inner classes for Person and AddressBook. Builders are also implemented for you. The generated classes implement com.google.protobuf.Message which is required by the serialisation mechanism. For this reason it important that only these classes are used in the body of your exchanges. Camel will throw an exception on route creation if you attempt to tell the Data Format to use a class that does not implement com.google.protobuf.Message. Use the generated builders to translate the data from any of your existing domain classes.

JAVA DSL

You can use create the ProtobufDataFormat instance and pass it to Camel DataFormat marshal and unmarshal API like this.

```
ProtobufDataFormat format = new ProtobufDataFormat(Person.getDefaultInstance());
from("direct:in").marshal(format);
from("direct:back").unmarshal(format).to("mock:reverse");
```

Or use the DSL protobuf() passing the unmarshal default instance or default instance class name like this.

```
// You don't need to specify the default instance for protobuf marshaling
from("direct:marshal").marshal().protobuf();
from("direct:unmarshalA").unmarshal().
    protobuf("org.apache.camel.dataformat.protobuf.generated.AddressBookProtos$Person").
    to("mock:reverse");

from("direct:unmarshalB").unmarshal().protobuf(Person.getDefaultInstance()).to("mock:reverse");
```

SPRING DSL

The following example shows how to use Castor to unmarshal using Spring configuring the protobuf data type

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <marshal>
            <protobuf instanceClass="org.apache.camel.dataformat.protobuf.generated.AddressBookProtos$Person" />
        </marshal>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

Dependencies

To use Protobuf in your camel routes you need to add the a dependency on **camel-protobuf** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-protobuf</artifactId>
  <version>2.2.0</version>
</dependency>
```

SOAP DATAFORMAT

Available as of Camel 2.3

SOAP is a Data Format which uses JAXB2 and JAX-WS annotations to marshal and unmarshal SOAP payloads. It provides the basic features of Apache CXF without need for the CXF Stack.

Supported SOAP versions

SOAP 1.1 is supported by default. SOAP 1.2 is supported from Camel 2.11 onwards.

Namespace prefix mapping

See JAXB for details how you can control namespace prefix mappings when marshalling using SOAP data format.

ElementNameStrategy

An element name strategy is used for two purposes. The first is to find a xml element name for a given object and soap action when marshaling the object into a SOAP message. The second is to find an Exception class for a given soap fault name.

Strategy	Usage
QNameStrategy	Uses a fixed qName that is configured on instantiation. Exception lookup is not supported
TypeNameStrategy	Uses the name and namespace from the @XMLType annotation of the given type. If no namespace is set then package-info is used. Exception lookup is not supported
ServiceInterfaceStrategy	Uses information from a webservice interface to determine the type name and to find the exception class for a SOAP fault

If you have generated the web service stub code with cxf-codegen or a similar tool then you probably will want to use the ServiceInterfaceStrategy. In the case you have no annotated service interface you should use QNameStrategy or TypeNameStrategy.

Using the Java DSL

The following example uses a named DataFormat of soap which is configured with the package com.example.customerService to initialize the JAXBContext. The second parameter is the ElementNameStrategy. The route is able to marshal normal objects as well as exceptions. (Note the below just sends a SOAP Envelope to a queue. A web service provider would actually need to be listening to the queue for a SOAP call to actually occur, in which case it would be a one way SOAP request. If you need request reply then you should look at the next example.)

```
SoapJaxbDataFormat soap = new SoapJaxbDataFormat("com.example.customerService", new ServiceInterfaceStrategy(CustomerService.class));
from("direct:start")
.marshal(soap)
.to("jms:myQueue");
```

See also

As the SOAP dataformat inherits from the JAXB dataformat most settings apply here as well

Using SOAP 1.2

Available as of Camel 2.11

```
SoapJaxbDataFormat soap = new SoapJaxbDataFormat("com.example.customerService", new ServiceInterfaceStrategy(CustomerService.class));
soap.setVersion("1.2");
from("direct:start")
.marshal(soap)
.to("jms:myQueue");
```

When using XML DSL there is a version attribute you can set on the <soapjaxb> element.

```
<!-- Defining a ServiceInterfaceStrategy for retrieving the element name when marshalling -->
<bean id="myNameStrategy" class="org.apache.camel.dataformat.soap.name.ServiceInterfaceStrategy">
  <constructor-arg value="com.example.customerService.CustomerService"/>
  <constructor-arg value="true"/>
</bean>
```

And in the Camel route

```
<route>
  <from uri="direct:start"/>
  <marshal>
    <soapjaxb contentPath="com.example.customerService" version="1.2" elementNameStrategyRef="myNameStrategy"/>
  </marshal>
  <to uri="jms:myQueue"/>
</route>
```

```
</route>
```

Multi-part Messages

Available as of Camel 2.8.1

Multi-part SOAP messages are supported by the ServiceInterfaceStrategy. The ServiceInterfaceStrategy must be initialized with a service interface definition that is annotated in accordance with JAX-WS 2.2 and meets the requirements of the Document Bare style. The target method must meet the following criteria, as per the JAX-WS specification: 1) it must have at most one in or in/out non-header parameter, 2) if it has a return type other than void it must have no in/out or out non-header parameters, 3) if it has a return type of void it must have at most one in/out or out non-header parameter.

The ServiceInterfaceStrategy should be initialized with a boolean parameter that indicates whether the mapping strategy applies to the request parameters or response parameters.

```
ServiceInterfaceStrategy strat = new ServiceInterfaceStrategy(com.example.customerservice.multipart.MultiPartCustomerService.class, true);
SoapJaxbDataFormat soapDataFormat = new SoapJaxbDataFormat("com.example.customerservice.multipart", strat);
```

Multi-part Request

The payload parameters for a multi-part request are initialized using a BeanInvocation object that reflects the signature of the target operation. The camel-soap DataFormat maps the content in the BeanInvocation to fields in the SOAP header and body in accordance with the JAX-WS mapping when the marshal() processor is invoked.

```
BeanInvocation beanInvocation = new BeanInvocation();
// Identify the target method
beanInvocation.setMethod(MultiPartCustomerService.class.getMethod("getCustomersByName",
    GetCustomersByName.class, com.example.customerservice.multipart.Product.class));
// Populate the method arguments
GetCustomersByName getCustomersByName = new GetCustomersByName();
getCustomersByName.setName("Dr. Multipart");
Product product = new Product();
product.setName("Multiuse Product");
product.setDescription("Useful for lots of things.");
Object[] args = new Object[] {getCustomersByName, product};
// Add the arguments to the bean invocation
beanInvocation.setArgs(args);
// Set the bean invocation object as the message body
exchange.getIn().setBody(beanInvocation);
```

Multi-part Response

A multi-part soap response may include an element in the soap body and will have one or more elements in the soap header. The camel-soap DataFormat will unmarshal the element in the soap body (if it exists) and place it onto the body of the out message in the exchange. Header elements will **not** be marshaled into their JAXB mapped object types. Instead, these elements are placed into the camel out message header org.apache.camel.dataformat.soap.UNMARSHALLED_HEADER_LIST. The elements will appear either as element instance values, or as JAXBElement values, depending upon the setting for the ignoreJAXBElement property. This property is inherited from camel-jaxb.

You can also have the camel-soap DataFormat ignore header content all-together by setting the ignoreUnmarshalledHeaders value to true.

Holder Object mapping

JAX-WS specifies the use of a type-parameterized javax.xml.ws.Holder object for in/out and out parameters. A Holder object may be used when building the BeanInvocation, or you may use an instance of the parameterized-type directly. The camel-soap DataFormat marshals Holder values in accordance with the JAXB mapping for the class of the Holder's value. No mapping is provided for Holder objects in an unmarshalled response.

Examples

Webservice client

The following route supports marshalling the request and unmarshalling a response or fault.

```
String WS_URI = "cxf://http://myserver/customerservice?serviceClass=com.example.customerservice&dataFormat=MESSAGE";
SoapJaxbDataFormat soapDF = new SoapJaxbDataFormat("com.example.customerservice", new ServiceInterfaceStrategy(CustomerService.class));
from("direct:customerServiceClient")
.onException(Exception.class)
.handled(true)
.unmarshal(soapDF)
.end()
.marshal(soapDF)
```

```
.to(ws_URI)
.unmarshal soapDF;
```

The below snippet creates a proxy for the service interface and makes a SOAP call to the above route.

```
import org.apache.camel.Endpoint;
import org.apache.camel.component.bean.ProxyHelper;
...
Endpoint startEndpoint = context.getEndpoint("direct:customerServiceClient");
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
// CustomerService below is the service endpoint interface, *not* the javax.xml.ws.Service subclass
CustomerService proxy = ProxyHelper.createProxy(startEndpoint, classLoader, CustomerService.class);
GetCustomersByNameResponse response = proxy.getCustomersByName(new GetCustomersByName());
```

Webservice Server

Using the following route sets up a webservice server that listens on jms queue customerServiceQueue and processes requests using the class CustomerServiceImpl. The customerServiceImpl of course should implement the interface CustomerService. Instead of directly instantiating the server class it could be defined in a spring context as a regular bean.

```
SoapJaxbDataFormat soapDF = new SoapJaxbDataFormat("com.example.customerservice", new ServiceInterfaceStrategy(CustomerService.class));
CustomerService serverBean = new CustomerServiceImpl();
from("jms://queue:customerServiceQueue")
.onException(Exception.class)
.handled(true)
.marshal(soapDF)
.end()
.unmarshal(soapDF)
.bean(serverBean)
.marshal(soapDF);
```

Dependencies

To use the SOAP dataformat in your camel routes you need to add the following dependency to your pom.

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-soap</artifactId>
<version>2.3.0</version>
</dependency>
```

CRYPTO

Available as of Camel 2.3

PGP Available as of Camel 2.9

The Crypto Data Format integrates the Java Cryptographic Extension into Camel, allowing simple and flexible encryption and decryption of messages using Camel's familiar marshall and unmarshal formatting mechanism. It assumes marshalling to mean encryption to ciphertext and unmarshalling to mean decryption back to the original plaintext. This data format implements only symmetric (shared-key) encryption and decryption.

Options

Name	Type	Default	Description
algorithm	string	DES/CBC/PKCS5Padding	The JCE algorithm name indicating the cryptographic algorithm that will be used.
algorithmParameterSpec	java.security.spec.AlgorithmParameterSpec	null	A JCE AlgorithmParameterSpec used to initialize the Cipher.
bufferSize	Integer	4096	the size of the buffer used in the signature process.
cryptoProvider	string	null	The name of the JCE Security Provider that should be used.
initializationvector	byte[]	null	A byte array containing the Initialization Vector that will be used to initialize the Cipher.

inline	boolean	false	Flag indicating that t configured IV should inlined into the encry data stream.
macAlgorithm	String	null	The JCE algorithm n indicating the Messa Authentication algori
shouldAppendHMAC	boolean	null	Flag indicating that a Message Authentica Code should be calculated and appe to the encrypted dat

Basic Usage

At its most basic all that is required to encrypt/decrypt an exchange is a shared secret key. If one or more instances of the Crypto data format are configured with this key the format can be used to encrypt the payload in one route (or part of one) and decrypted in another. For example, using the Java DSL as follows:{snippet:id=basic|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}In Spring the dataformat is configured first and then used in routes

```
xml<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring"> <dataFormats> <crypto id="basic" algorithm="DES" keyRef="desKey" /> </dataFormats> ... <route> <from uri="direct:basic-encryption" /> <marshal ref="basic" /> <to uri="mock:encrypted" /> <unmarshal ref="basic" /> <to uri="mock:unencrypted" /> </route> </camelContext>
```

Specifying the Encryption Algorithm

Changing the algorithm is a matter of supplying the JCE algorithm name. If you change the algorithm you will need to use a compatible key.{snippet:id=algorithm|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}A list of the available algorithms in Java 7 is available via the Java Cryptography Architecture Standard Algorithm Name Documentation.

Specifying an Initialization Vector

Some crypto algorithms, particularly block algorithms, require configuration with an initial block of data known as an Initialization Vector. In the JCE this is passed as an AlgorithmParameterSpec when the Cipher is initialized. To use such a vector with the CryptoDataFormat you can configure it with a byte[] containing the required data e.g.{snippet:id=init-vector|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}or with spring, suppling a reference to a byte[] {snippet:id=init-vector|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringCryptoDataFormatTest.xml}The same vector is required in both the encryption and decryption phases. As it is not necessary to keep the IV a secret, the DataFormat allows for it to be inlined into the encrypted data and subsequently read out in the decryption phase to initialize the Cipher. To inline the IV set the /oinline flag.{snippet:id=inline-init-vector|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}or with spring.{snippet:id=inline|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringCryptoDataFormatTest.xml}For more information of the use of Initialization Vectors, consult

- http://en.wikipedia.org/wiki/Initialization_vector
- <http://www.herongyang.com/Cryptography/>
- http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

Hashed Message Authentication Codes (HMAC)

To avoid attacks against the encrypted data while it is in transit the CryptoDataFormat can also calculate a Message Authentication Code for the encrypted exchange contents based on a configurable MAC algorithm. The calculated HMAC is appended to the stream after encryption. It is separated from the stream in the decryption phase. The MAC is recalculated and verified against the transmitted version to insure nothing was tampered with in transit. For more information on Message Authentication Codes see [{snippet:id=hmac|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}](http://en.wikipedia.org/wiki/HMAC)or with spring.{snippet:id=hmac-algorithm|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringCryptoDataFormatTest.xml}By default the HMAC is calculated using the HmacSHA1 mac algorithm though this can be easily changed by supplying a different algorithm name. See here for how to check what algorithms are available through the configured security providers{snippet:id=hmac-algorithm|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}or with spring.{snippet:id=hmac-algorithm|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringCryptoDataFormatTest.xml}

Supplying Keys Dynamically

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

- `CryptoDataFormat.KEY "CamelCryptokey"`

```
{snippet:id=key-in-header|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java} or with spring.{snippet:id=header-key|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringCryptoDataFormatTest.xml}
```

PGP Message

The PGP Data Formater can create and decrypt/verify PGP Messages of the following PGP packet structure (entries in brackets are optional and ellipses indicate repetition, comma represents sequential composition, and vertical bar separates alternatives):

Public Key Encrypted Session Key ..., Symmetrically Encrypted Data | Sym. Encrypted and Integrity Protected Data, (Compressed Data,) (One Pass Signature ...,) Literal Data, (Signature ...)

Since Camel 2.16.0 the Compressed Data packet is optional, before it was mandatory.

PGPDataFormat Options

Name	Type	Default	Description
keyuserid	String	null	The user ID of the key in be only a part of a user ID the part "Test User" or <t
keyuserids	List<String>	null	Since camel 2.12.2: PGF You can specify here the If you just have one User keyuserid and the User be used for the encryptio
password	String	null	Password used when ope
keyFileName	String	null	Filename of the keyring; r file system by using the "i
encryptionKeyRing	byte[]	null	Since camel 2.12.1: enc same time.
signatureKeyuserid	String	null	Since Camel 2.11.0: opti signature verification (dur restricts the public keys fr specified for the signature verification. Can also be c <test@camel.com>" then
signatureKeyuserids	List<String>	null	Since Camel 2.12.3: opti encryption) or signature v IDs of several keys conta option keyuserid. The U together and the correspo User IDs reference sever encryption-signing proces keys which can be used f public keyring can be use
signaturePassword	String	null	Since Camel 2.11.0: opti encryption).
signatureKeyFileName	String	null	Since Camel 2.11.0: opti verification (during decry in the file system by using
signatureKeyRing	byte[]	null	Since camel 2.12.1: sign at the same time.
algorithm	int	SymmetricKeyAlgorithmTags.CAST5	Since camel 2.12.2: sym org.bouncycastle.bcpc (= BLOWFISH), 6 (= DES
compressionAlgorithm	int	CompressionAlgorithmTags.ZIP	Since camel 2.12.2: com

			org.bouncycastle.bcp 2 (= ZLIB), 3 (= BZIP2). C
hashAlgorithm	int	HashAlgorithmTags.SHA1	Since camel 2.12.2: sign org.bouncycastle.bcp (= SHA512), 11 (=SHA22
armored	boolean	false	This option will cause PG
integrity	boolean	true	Adds an integrity check/s
passphraseAccessor	PGPPassphraseAccessor	null	Since Camel 2.12.2: prov from the option password CamelPGPDataFormatKey is fetched from the passp PGPPassphraseAccesso passphrase accessor is e Messages Encrypted/Sig
signatureVerificationOption	String	"optional"	Since Camel 2.13.0: con values possible: <ul style="list-style-type: none">• "optional": The f then a signature ve PGPKeyAccessDa• "required": The f exception (PGPEx PGPKeyAccessDa• "ignore": Contain executed. Use the PGPKeyAccessDa• "no_signature_a exception (PGPEx PGPKeyAccessDa
FileName	String	"_CONSOLE"	Since camel 2.15.0: Sets Exchange#FILE_NAME}. "_CONSOLE" indicates t message data is unusuall avoiding storing the recei
withCompressedDataPacket	boolean	true	Since Camel 2.16.0: Indi Data packet. If the value i compressionAlgorithm va

PGPDataFormat Message Headers

You can override the PGPDataFormat options by applying below headers into message dynamically.

Name	Type	Description
CamelPGPDataFormatKeyFileName	String	Since Camel 2.11.0: filename of the keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatEncryptionKeyRing	byte[]	Since Camel 2.12.1: the encryption keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatKeyuserid	String	Since Camel 2.11.0: the User ID of the key in the PGP keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatKeyuserids	List<String>	Since camel 2.12.2: the User IDs of the key in the PGP keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatKeyPassword	String	Since Camel 2.11.0: password used when opening the private key; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatSignatureKeyFileName	String	Since Camel 2.11.0: filename of the signature keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatSignatureKeyRing	byte[]	Since Camel 2.12.1: the signature keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatSignatureKeyuserid	String	Since Camel 2.11.0: the User ID of the signature key in the PGP keyring; will override existing setting directly on the PGPDataFormat.

<code>CamelPGPDataFormatSignatureKeyUserids</code>	<code>List<String></code>	Since Camel 2.12.3: the User IDs of the signature keys in the PGP keyring; will override existing setting directly on the PGPDataFormat.
<code>CamelPGPDataFormatSignatureKeyPassword</code>	<code>String</code>	Since Camel 2.11.0: password used when opening the signature private key; will override existing setting directly on the PGPDataFormat.
<code>CamelPGPDataFormatEncryptionAlgorithm</code>	<code>int</code>	Since Camel 2.12.2: symmetric key encryption algorithm; will override existing setting directly on the PGPDataFormat.
<code>CamelPGPDataFormatSignatureHashAlgorithm</code>	<code>int</code>	Since Camel 2.12.2: signature hash algorithm; will override existing setting directly on the PGPDataFormat.
<code>CamelPGPDataFormatCompressionAlgorithm</code>	<code>int</code>	Since Camel 2.12.2: compression algorithm; will override existing setting directly on the PGPDataFormat.
<code>CamelPGPDataFormatNumberOfEncryptionKeys</code>	<code>Integer</code>	Since Camel 2.12.3: number of public keys used for encrypting the symmetric key, set by PGPDataFormat during encryption process
<code>CamelPGPDataFormatNumberofSigningKeys</code>	<code>Integer</code>	Since Camel 2.12.3: number of private keys used for creating signatures, set by PGPDataFormat during signing process

Encrypting with PGPDataFormat

The following sample uses the popular PGP format for encrypting/decrypting files using the Bouncy Castle Java libraries:
{snippet:id=pgp-format|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/PGPDataFormatTest.java}The following sample performs signing + encryption, and then signature verification + decryption. It uses the same keyring for both signing and encryption, but you can obviously use different keys:{snippet:id=pgp-format-signature|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/PGPDataFormatTest.java}Or using Spring:{snippet:id=pgp-xml-basic|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringPGPDataFormatTest.xml}

To work with the previous example you need the following

- A public keyring file which contains the public keys used to encrypt the data
- A private keyring file which contains the keys used to decrypt the data
- The keyring password

Managing your keyring

To manage the keyring, I use the command line tools, I find this to be the simplest approach in managing the keys. There are also Java libraries available from <http://www.bouncycastle.org/java.html> if you would prefer to do it that way.

1. Install the command line utilities on linux
`apt-get install gnupg`
2. Create your keyring, entering a secure password
`gpg --gen-key`
3. If you need to import someone elses public key so that you can encrypt a file for them.
`gpg --import <filename.key`
4. The following files should now exist and can be used to run the example
`ls -l ~/gnupg/pubring.gpg ~/gnupg/secring.gpg`

PGP Decrypting/Verifying of Messages Encrypted/Signed by Different Private/Public Keys

Since Camel 2.12.2.

A PGP Data Formater can decrypt/verify messages which have been encrypted by different public keys or signed by different private keys. Just, provide the corresponding private keys in the secret keyring, the corresponding public keys in the public keyring, and the passphrases in the passphrase accessor.

```
javaMap<String, String> userId2Passphrase = new HashMap<String, String>(2); // add passphrases of several private keys whose corresponding public keys have been used to encrypt the messages
userId2Passphrase.put("UserIdOfKey1", "passphrase1"); // you must specify the exact User ID!
userId2Passphrase.put("UserIdOfKey2", "passphrase2");
PGPPassphraseAccessor passphraseAccessor = new PGPPassphraseAccessorDefault(userId2Passphrase);
PGPDataFormat pgpVerifyAndDecrypt = new PGPDataFormat();
pgpVerifyAndDecrypt.setPassphraseAccessor(passphraseAccessor); // the method getSecKeyRing() provides the secret keyring as byte array containing the private keys
pgpVerifyAndDecrypt.setEncryptionKeyRing(getSecKeyRing()); // alternatively you can use setKeyFileName(keyfileName) // the method getPublicKeyRing() provides the public keyring as byte array containing the public keys
pgpVerifyAndDecrypt.setSignatureKeyRing((getPublicKeyRing())); // alternatively you can use
```

```
setSignatureKeyFileName(signatgureKeyfileName) // it is not necessary to specify the encryption or signer User Id
from("direct:start") ... .unmarshal(ppgVerifyAndDecrypt) // can decrypt/verify messages encrypted/signed by different
private/public keys ...
```

- The functionality is especially useful to support the key exchange. If you want to exchange the private key for decrypting you can accept for a period of time messages which are either encrypted with the old or new corresponding public key. Or if the sender wants to exchange his signer private key, you can accept for a period of time, the old or new signer key.
- Technical background: The PGP encrypted data contains a Key ID of the public key which was used to encrypt the data. This Key ID can be used to locate the private key in the secret keyring to decrypt the data. The same mechanism is also used to locate the public key for verifying a signature. Therefore you no longer must specify User IDs for the unmarshaling.

Restricting the Signer Identities during PGP Signature Verification

Since Camel 2.12.3.

If you verify a signature you not only want to verify the correctness of the signature but you also want check that the signature comes from a certain identity or a specific set of identities. Therefore it is possible to restrict the number of public keys from the public keyring which can be used for the verification of a signature.

```
javaSignature User IDs// specify the User IDs of the expected signer identities List<String> expectedSigUserIds = new
ArrayList<String>(); expectedSigUserIds.add("Trusted company1"); expectedSigUserIds.add("Trusted company2");
PGPDataFormat ppgVerifyWithSpecificKeysAndDecrypt = new PGPDataFormat();
ppgVerifyWithSpecificKeysAndDecrypt.setPassword("my password"); // for decrypting with private key
ppgVerifyWithSpecificKeysAndDecrypt.setKeyFileName(keyfileName);
ppgVerifyWithSpecificKeysAndDecrypt.setSignatureKeyFileName(signatgureKeyfileName);
ppgVerifyWithSpecificKeysAndDecrypt.setSignatureKeyUserids(expectedSigUserIds); // if you have only one signer identity then
you can also use setSignatureKeyUserid("expected Signer") from("direct:start") ...
.unmarshal(ppgVerifyWithSpecificKeysAndDecrypt) ...
```

- If the PGP content has several signatures the verification is successful as soon as one signature can be verified.
- If you do not want to restrict the signer identities for verification then do not specify the signature key User IDs. In this case all public keys in the public keyring are taken into account.

Several Signatures in One PGP Data Format

Since Camel 2.12.3.

The PGP specification allows that one PGP data format can contain several signatures from different keys. Since Camel 2.13.3 it is possible to create such kind of PGP content via specifying signature User IDs which relate to several private keys in the secret keyring.

```
javaSeveral Signatures PGPDataFormat ppgSignAndEncryptSeveralSignerKeys = new PGPDataFormat();
ppgSignAndEncryptSeveralSignerKeys.setKeyUserid(keyUserid); // for encrypting, you can also use setKeyUserids if you want to
encrypt with several keys ppgSignAndEncryptSeveralSignerKeys.setKeyFileName(keyfileName);
ppgSignAndEncryptSeveralSignerKeys.setSignatureKeyFileName(signatgureKeyfileName);
ppgSignAndEncryptSeveralSignerKeys.setSignaturePassword("sdude"); // here we assume that all private keys have the same
password, if this is not the case then you can use setPassphraseAccessor List<String> signerUserIds = new ArrayList<String>();
signerUserIds.add("company old key"); signerUserIds.add("company new key");
ppgSignAndEncryptSeveralSignerKeys.setSignatureKeyUserids(signerUserIds); from("direct:start") ...
.marshall(ppgSignAndEncryptSeveralSignerKeys) ...
```

Support of Sub-Keys and Key Flags in PGP Data Format Marshaler

Since Camel 2.12.3.

An OpenPGP V4 key can have a primary key and sub-keys. The usage of the keys is indicated by the so called Key Flags. For example, you can have a primary key with two sub-keys; the primary key shall only be used for certifying other keys (Key Flag 0x01), the first sub-key shall only be used for signing (Key Flag 0x02), and the second sub-key shall only be used for encryption (Key Flag 0x04 or 0x08). The PGP Data Format marshaler takes into account these Key Flags of the primary key and sub-keys in order to determine the right key for signing and encryption. This is necessary because the primary key and its sub-keys have the same User IDs.

Support of Custom Key Accessors

Since Camel 2.13.0.

You can implement custom key accessors for encryption/signing. The above PGPDataFormat class selects in a certain predefined way the keys which should be used for signing/encryption or verifying/decryption. If you have special requirements how your keys should be selected you should use the PGPKAccessDataFormat class instead and implement the interfaces PGPPublicKeyAccessor and PGPSecretKeyAccessor as beans. There are default implementations

DefaultPGPPublicKeyAccessor and DefaultPGPSecretKeyAccessor which cache the keys, so that not every time the keyring is parsed when the processor is called.

PGPKAccessDataFormat has the same options as PGPDataFormat except password, keyFileName, encryptionKeyRing, signaturePassword, signatureKeyFileName, and signatureKeyRing.

Dependencies

To use the Crypto dataformat in your camel routes you need to add the following dependency to your pom.

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

See Also

- Data Format
- Crypto (Digital Signatures)
- <http://www.bouncycastle.org/java.html>

SYSLOG DATAFORMAT

Available as of Camel 2.6

The **syslog** dataformat is used for working with RFC3164 and RFC5424 messages.

This component supports the following:

- UDP consumption of syslog messages
- Agnostic data format using either plain String objects or SyslogMessage model objects.
- Type Converter from/to SyslogMessage and String
- Integration with the camel-mina component.
- Integration with the camel-netty component.
- **Camel 2.14:** Encoder and decoder for the camel-netty component.
- **Camel 2.14:** Support for RFC5424 also.

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

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

RFC3164 Syslog protocol

Syslog uses the user datagram protocol (UDP) 1 as its underlying transport layer mechanism.
The UDP port that has been assigned to syslog is 514.

To expose a Syslog listener service we reuse the existing camel-mina component or camel-netty where we just use the `Rfc3164SyslogDataFormat` to marshal and unmarshal messages. Notice that from **Camel 2.14** onwards the syslog dataformat is renamed to `SyslogDataFormat`.

RFC5424 Syslog protocol

Available as of Camel 2.14

To expose a Syslog listener service we reuse the existing camel-mina component or camel-netty where we just use the `SyslogDataFormat` to marshal and unmarshal messages

Exposing a Syslog listener

In our Spring XML file, we configure an endpoint to listen for udp messages on port 10514, note that in netty we disable the `defaultCodec`, this will allow a fallback to a NettyTypeConverter and delivers the message as an `InputStream`:

```
<camelContext id="myCamel" xmlns="http://camel.apache.org/schema/spring">
  <dataFormats>
    <syslog id="mySyslog"/>
  </dataFormats>

  <route>
    <from uri="netty:udp://localhost:10514?sync=false&allowDefaultCodec=false"/>
    <unmarshal ref="mySyslog"/>
    <to uri="mock:stop1"/>
  </route>
</camelContext>
```

The same route using camel-mina

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

```
<syslog id="mySyslog"/>
</dataFormats>
<route>
<from uri="mina:udp://localhost:10514"/>
<unmarshal ref="mySyslog"/>
<to uri="mock:stop1"/>
</route>
</camelContext>
```

Sending syslog messages to a remote destination

```
<camelContext id="myCamel" xmlns="http://camel.apache.org/schema/spring">
<dataFormats>
<syslog id="mySyslog"/>
</dataFormats>
<route>
<from uri="direct:syslogMessages"/>
<marshal ref="mySyslog"/>
<to uri="mina:udp://remotehost:10514"/>
</route>
</camelContext>
```

See Also

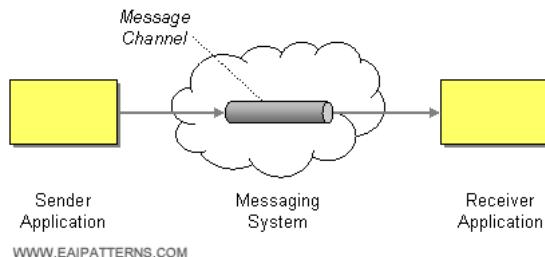
- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 10**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.

**Example**

In JMS, Message Channels are represented by topics and queues such as the following

```
jms:queue:foo
```

This message channel can be then used within the JMS component

Using the Fluent Builders

```
to("jms:queue:foo")
```

Using the Spring XML Extensions

```
<to uri="jms:queue:foo"/>
```

For more details see

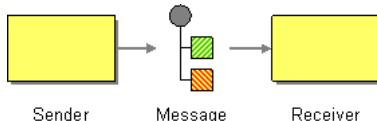
- Message
- Message Endpoint

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 Message and Request Reply messages Camel uses an Exchange interface which has a **pattern** property which can be set to **InOnly** for an Event Message which has a single inbound Message, or **InOut** for a Request Reply where there is an inbound and outbound message.

Here is a basic example of sending a Message to a route in **InOnly** and **InOut** modes

Requestor Code

```
//InOnly
getContext().createProducerTemplate().sendBody("direct:startInOnly", "Hello World");

//InOut
String result = (String) getContext().createProducerTemplate().requestBody("direct:startInOut", "Hello World");
```

Route Using the Fluent Builders

```
from("direct:startInOnly").inOnly("bean:process");
from("direct:startInOut").inOut("bean:process");
```

Route Using the Spring XML Extensions

```
<route>
<from uri="direct:startInOnly"/>
<inOnly uri="bean:process"/>
</route>

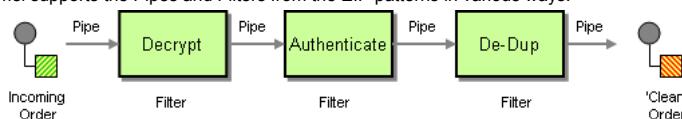
<route>
<from uri="direct:startInOut"/>
<inOut uri="bean:process"/>
</route>
```

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.

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{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/PipelineTest.java}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

```
xml: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:

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

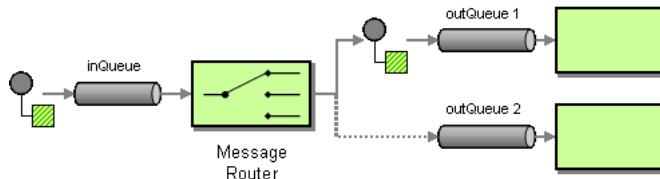
```
xml:route> <from uri="activemq:SomeQueue"/> <multicast> <pipeline> <bean ref="something"/> <to uri="log:Something"/> </pipeline> <pipeline> <bean ref="foo"/> <bean ref="bar"/> <to uri="activemq:OutputQueue"/> </pipeline> </multicast> </route>
```

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

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() {
        errorHandler(mockErrorHandler("errorHandler"));

        from("direct:a")
            .choice()
                .when(header("foo").isEqualTo("bar"))
                    .to("direct:b")
                .when(header("foo").isEqualTo("cheese"))
                    .to("direct:c")
                .otherwise()
                    .to("direct:d");
    }
};
  
```

Using the Spring XML Extensions

```

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

Choice without otherwise

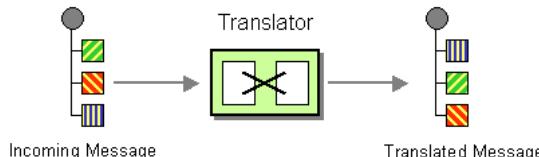
If you use a choice without adding an otherwise, any unmatched exchanges will be dropped by default.

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

```
{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/TransformTest.java}
```

or you can use the DSL to explicitly configure the transformation

```
{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/TransformProcessorTest.java}
```

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

```
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/transformWithExpressionContext.xml}
```

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");
```

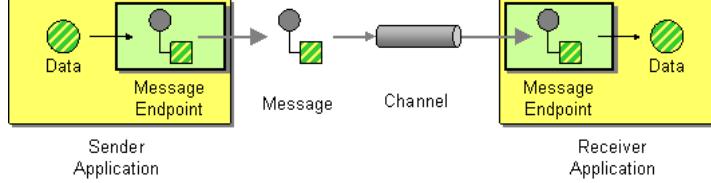
If you want to use InOut (request-reply) semantics to process requests on the **My.Queue** queue on ActiveMQ with a template generated response, then sending responses back to the JMSReplyTo Destination you could use this.

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

- Content Enricher
- Using getIn or getOut methods on Exchange

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.

Example

The following example route demonstrates the use of a File Consumer Endpoint and JMS Producer Endpoint

Using the Fluent Builders

```
from("file://local/router/messages/foo")
    .to("jms:queue:foo");
```

Using the Spring XML Extensions

```
<route>
    <from uri="file://local/router/messages/foo"/>
    <to uri="jms:queue:foo"/>
</route>
```

Dynamic To

Available as of Camel 2.16

There is a new <toD> that allows to send a message to a dynamic computed Endpoint using one or more Expression that are concat together. By default the Simple language is used to compute the endpoint. For example to send a message to a endpoint defined by a header you can do

```
<route>
    <from uri="direct:start"/>
    <toD uri="#${header.foo}"/>
</route>
```

And in Java DSL

```
from("direct:start")
    .toD("${header.foo}");
```

You can also prefix the uri with a value because by default the uri is evaluated using the Simple language

```
<route>
    <from uri="direct:start"/>
    <toD uri="mock:${header.foo}"/>
</route>
```

And in Java DSL

```
from("direct:start")
    .toD("mock:${header.foo}");
```

In the example above we compute an endpoint that has prefix "mock:" and then the header foo is appended. So for example if the header foo has value order, then the endpoint is computed as "mock:order".

You can also use other languages than Simple such as XPath - this requires to prefix with language: as shown below (simple language is the default language). If you do not specify language: then the endpoint is a component name. And in some cases there is both a component and language with the same name such as xquery.

```
<route>
    <from uri="direct:start"/>
    <toD uri="language:xpath:/order/@uri"/>
</route>
```

This is done by specifying the name of the language followed by a colon.

```
from("direct:start")
    .toD("language:xpath:/order/@uri");
```

You can also concat multiple Language(s) together using the plus sign + such as shown below:

```
<route>
    <from uri="direct:start"/>
    <toD uri="jms:${header.base}+language:xpath:/order/@id"/>
</route>
```

In the example above the uri is a combination of Simple language and XPath where the first part is simple (simple is default language). And then the plus sign separate to another language, where we specify the language name followed by a colon

```
from("direct:start")
    .toD("jms:${header.base}+language:xpath:/order/@id");
```

You can concat as many languages as you want, just separate them with the plus sign

The Dynamic To has a few options you can configure

Name	Default Value	Description
uri		Mandatory: The uri to use. See above
pattern		To set a specific Exchange Pattern to use when sending to the endpoint. The original MEP is restored afterwards.
cacheSize		Allows to configure the cache size for the ProducerCache which caches producers for reuse. Will by default use the default cache size which is 1000. Setting the value to -1 allows to turn off the cache all together.
ignoreInvalidEndpoint	false	Whether to ignore an endpoint URI that could not be resolved. If disabled, Camel will throw an exception identifying the invalid endpoint URI.

For more details see

- Recipient List
- Message
- Wire Tap

Using This Pattern

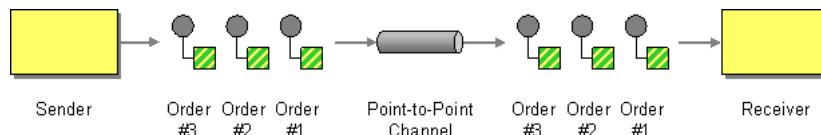
If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

MESSAGING CHANNELS

Point to Point Channel

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

- SEDA 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)
- and others



The following example demonstrates point to point messaging using the JMS component

Using the Fluent Builders

```
from("direct:start")
    .to("jms:queue:foo");
```

Using the Spring XML Extensions

```
<route>
    <from uri="direct:start"/>
    <to uri="jms:queue:foo"/>
</route>
```

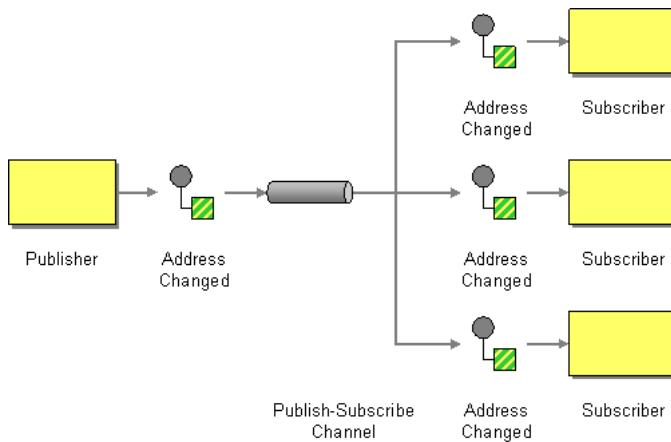
Using This Pattern

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.

Publish Subscribe Channel

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

- JMS for working with JMS Topics for high performance, clustering and load balancing
- XMPP when using rooms for group communication
- SEDA for working with SEDA in the same CamelContext which can work in pub-sub, but allowing multiple consumers.
- VM as SEDA but for intra-JVM.



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() {
        errorHandler(deadLetterChannel("mock:error"));

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

Using the Spring XML Extensions

```

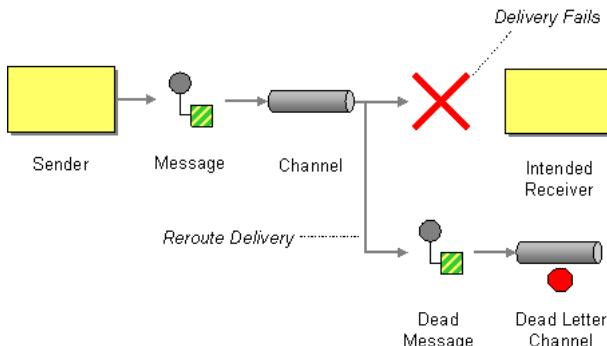
<camelContext errorHandlerRef="errorHandler" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:a"/>
        <multicast>
            <to uri="direct:b"/>
            <to uri="direct:c"/>
            <to uri="direct:d"/>
        </multicast>
    </route>
</camelContext>
  
```

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.



Differences Between The DeadLetterChannel And The DefaultErrorHandler

The `defaultErrorHandler` does very little: it ends the Exchange immediately and propagates the thrown Exception back to the caller.

The `deadLetterChannel` lets you control behaviors including redelivery, whether to propagate the thrown Exception to the caller (the `handled` option), and where the (failed) Exchange should now be routed to.

The `deadLetterChannel` is also by default configured to not be verbose in the logs, so when a message is handled and moved to the dead letter endpoint, there is nothing logged. If you want some level of logging you can use the various options on the redelivery policy / dead letter channel to configure this. For example if you want the message history then set `logExhaustedMessageHistory=true` (and `logHandled=true` for Camel 2.15.x or older).

When the `deadLetterChannel` moves a message to the dead letter endpoint, any new Exception thrown is by default handled by the dead letter channel as well. This ensures that the `deadLetterChannel` will always succeed. From Camel 2.15: this behavior can be changed by setting the option `deadLetterHandleNewException=false`. Then if a new Exception is thrown, then the dead letter channel will fail and propagate back that new Exception (which is the behavior of the default error handler). When a new Exception occurs then the dead letter channel logs this at `WARN` level. This can be turned off by setting `logNewException=false`.

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

- The number of 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.
- Delay pattern (see below for details).
- Camel 2.11: Whether to allow redelivery during stopping/shutdown.

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

About Moving Exchange to Dead Letter Queue and Using `handled()`

`handled()` on Dead Letter Channel

When all attempts of redelivery have failed the Exchange is moved to the dead letter queue (the dead letter endpoint). The exchange is then complete and from the client point of view it was processed. As such the Dead Letter Channel have handled the Exchange.

For instance configuring the dead letter channel as:

Using the Fluent Builders

```
javaErrorHandler(deadLetterChannel("jms:queue:dead") .maximumRedeliveries(3).redeliveryDelay(5000));
```

Using the Spring XML Extensions

```
xml->route errorHandlerRef="myDeadLetterErrorHandler"><!-- ... --> </route> <bean id="myDeadLetterErrorHandler" class="org.apache.camel.builder.DeadLetterChannelBuilder"> <property name="deadLetterUri" value="jms:queue:dead"/> <property name="redeliveryPolicy" ref="myRedeliveryPolicyConfig"/> </bean> <bean id="myRedeliveryPolicyConfig" class="org.apache.camel.processor.RedeliveryPolicy"> <property name="maximumRedeliveries" value="3"/> <property name="redeliveryDelay" value="5000"/> </bean>
```

The Dead Letter Channel above will clear the caused exception `setException(null)`, by moving the caused exception to a property on the Exchange, with the key `Exchange.EXCEPTION_CAUGHT`. Then the Exchange is moved to the `jms:queue:dead` destination and the client will not notice the failure.

About Moving Exchange to Dead Letter Queue and Using the Original Message

The option `useOriginalMessage` is used for routing the original input message instead of the current message that potentially is modified during routing.

For instance if you have this route:
`javafrom("jms:queue:order:input").to("bean:validateOrder").to("bean:transformOrder").to("bean:handleOrder");`
The route listen for JMS messages and validates, transforms and handle it. During this the Exchange payload is transformed/modified. So in case something goes wrong and we want to move the message to another JMS destination, then we can configure our Dead Letter Channel with the `useOriginalMessage` option. But when we move the Exchange to this destination we do not know in which state the message is in. Did the error happen in before the `transformOrder` or after? So to be sure we want to move the original input message we received from `jms:queue:order:input`. So we can do this by enabling the `useOriginalMessage` option as shown below:
`java// will use original body errorHandler(deadLetterChannel("jms:queue:dead").useOriginalMessage().maximumRedeliveries(5).redeliveryDelay(5000);`
Then the messages routed to the `jms:queue:dead` is the original input. If we want to manually retry we can move the JMS message from the failed to the input queue, with no problem as the message is the same as the original we received.

OnRedelivery

When Dead Letter Channel is doing redeliver its possible to configure a Processor that is executed just **before** every redelivery attempt. This can be used for the situations where you need to alter the message before its redelivered. See below for sample. `onException` and `onRedelivery`

We also support for per `onException` to set an `onRedelivery`. That means you can do special on redelivery for different exceptions, as opposed to `onRedelivery` set on Dead Letter Channel can be viewed as a global scope.

Redelivery Default Values

Redelivery is disabled by default.

The default redeliver policy will use the following values:

- `maximumRedeliveries=0`
- `redeliveryDelay=1000L` (1 second)
- `maximumRedeliveryDelay = 60 * 1000L` (60 seconds)
- `backoffMultiplier` and `useExponentialBackoff` are ignored.
- `retriesExhaustedLogLevel=LoggingLevel.ERROR`
- `retryAttemptedLogLevel=LoggingLevel.DEBUG`
- Stack traces are logged for exhausted messages, from **Camel 2.2**.
- Handled exceptions are not logged, from **Camel 2.3**.
- `logExhaustedMessageHistory` is true for default error handler, and false for dead letter channel.
- `logExhaustedMessageBody` **Camel 2.17**: is disabled by default to avoid logging sensitive message body/header details. If this option is true, then `logExhaustedMessageHistory` must also be true.

The maximum redeliver delay ensures that a delay is never longer than the value, default 1 minute. This can happen when `useExponentialBackoff=true`.

The `maximumRedeliveries` is the number of re-delivery attempts. By default Camel will try to process the exchange 1 + 5 times. 1 time for the normal attempt and then 5 attempts as redeliveries.

Setting the `maximumRedeliveries=-1` (or `< -1`) will then always redelivery (unlimited).

Setting the `maximumRedeliveries=0` will disable re-delivery.

Camel will log delivery failures at the `DEBUG` logging level by default. You can change this by specifying `retriesExhaustedLogLevel` and/or `retryAttemptedLogLevel`. See `ExceptionBuilderWithRetryLoggingLevelSetTest` for an example.

You can turn logging of stack traces on/off. If turned off Camel will still log the redelivery attempt. It's just much less verbose.

Redeliver Delay Pattern

Delay pattern is used as a single option to set a range pattern for delays. When a delay pattern is in use the following options no longer apply:

- `delay`
- `backoffMultiplier`
- `useExponentialBackoff`
- `useCollisionAvoidance`
- `maximumRedeliveryDelay`

The idea is to set groups of ranges using the following syntax: `limit:delay;limit 2:delay 2;limit 3:delay 3;...;limit N:delay N`

Each group has two values separated with colon:

- `limit = upper limit`

- `delay` = delay in milliseconds
And the groups is again separated with semi-colon. The rule of thumb is that the next groups should have a higher limit than the previous group.

Lets clarify this with an example:
`delayPattern=5:1000;10:5000;20:20000`

That gives us three groups:

- 5:1000
- 10:5000
- 20:20000

Resulting in these delays between redelivery attempts:

- Redelivery attempt number 1..4 = 0ms (as the first group start with 5)
- Redelivery attempt number 5..9 = 1000ms (the first group)
- Redelivery attempt number 10..19 = 5000ms (the second group)
- Redelivery attempt number 20.. = 20000ms (the last group)

Note: The first redelivery attempt is 1, so the first group should start with 1 or higher.

You can start a group with limit 1 to e.g., have a starting delay: `delayPattern=1:1000;5:5000`

- Redelivery attempt number 1..4 = 1000ms (the first group)
- Redelivery attempt number 5.. = 5000ms (the last group)

There is no requirement that the next delay should be higher than the previous. You can use any delay value you like. For example with `delayPattern=1:5000;3:1000` we start with 5 sec delay and then later reduce that to 1 second.

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.

Before Camel 2.6: The header is `CamelRedeliveryCounter`, which is also defined on the Exchange.`REDELIVERY_COUNTER`.

From Camel 2.6: The header `CamelRedeliveryMaxCounter`, which is also defined on the Exchange.`REDELIVERY_MAX_COUNTER`, contains the maximum redelivery setting. This header is absent if you use `retryWhile` or have unlimited maximum redelivery configured.

And a boolean flag whether it is being redelivered or not (first attempt). The header `CamelRedelivered` contains a boolean if the message is redelivered or not, which is also defined on the Exchange.`REDELIVERED`.

Dynamically Calculated Delay From the Exchange

In Camel 2.9 and 2.8.2: The header is `CamelRedeliveryDelay`, which is also defined on the Exchange.`REDELIVERY_DELAY`. If this header is absent, normal redelivery rules apply.

Which Endpoint Failed

Available as of Camel 2.1

When Camel routes messages it will decorate the Exchange with a property that contains the `last` endpoint Camel send the Exchange to:

```
javaString lastEndpointUri = exchange.getProperty(Exchange.TO_ENDPOINT, String.class);
```

The Exchange.`TO_ENDPOINT` have the constant value `CamelToEndpoint`. This information is updated when Camel sends a message to any endpoint. So if it exists its the `last` endpoint which Camel send the Exchange to.

When for example processing the Exchange at a given Endpoint and the message is to be moved into the dead letter queue, then Camel also decorates the Exchange with another property that contains that `last` endpoint:

```
javaString failedEndpointUri = exchange.getProperty(Exchange.FAILURE_ENDPOINT, String.class);
```

The Exchange.`FAILURE_ENDPOINT` have the constant value `CamelFailureEndpoint`.

This allows for example you to fetch this information in your dead letter queue and use that for error reporting. This is usable if the Camel route is a bit dynamic such as the dynamic Recipient List so you know which endpoints failed.

Note: this information is retained on the Exchange even if the message is subsequently processed successfully by a given endpoint only to fail, for example, in local Bean processing instead. So, beware that this is a hint that helps pinpoint errors.

```
javaFrom("activemq:queue:foo").to("http://someserver/somepath").beanRef("foo");
```

Now suppose the route above and a failure happens in the foo bean. Then the Exchange.`TO_ENDPOINT` and Exchange.`FAILURE_ENDPOINT` will still contain the value of `http://someserver/somepath`.

OnPrepareFailure

Available as of Camel 2.16

Before the exchange is sent to the dead letter queue, you can use `onPrepare` to allow a custom Processor to prepare the exchange, such as adding information why the Exchange failed.

For example, the following processor adds a header with the exception message:

```
java public static class MyPrepareProcessor implements Processor { @Override public void process(Exchange exchange) throws Exception { Exception cause = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.class); exchange.getln().setHeader("FailedBecause", cause.getMessage()); } }
Then configure the error handler to use the processor as follows:
javaErrorHandler(deadLetterChannel("jms:dead").onPrepareFailure(new MyPrepareProcessor()));
```

Configuring this from XML DSL is as follows:

```
<xml>
<bean id="myPrepare" class="org.apache.camel.processor.DeadLetterChannelOnPrepareTest.MyPrepareProcessor"/>
<errorHandler id="dlc" type="DeadLetterChannel" deadLetterUri="jms:dead" onPrepareFailureRef="myPrepare"/>
```

The `onPrepare` is also available using the default error handler.

Which Route Failed

Available as of Camel 2.10.4/2.11

When Camel error handler handles an error such as Dead Letter Channel or using Exception Clause with `handled=true`, then Camel will decorate the Exchange with the route id where the error occurred.

Example:

```
javaString failedRouteId = exchange.getProperty(Exchange.FAILURE_ROUTE_ID, String.class);
The Exchange.FAILURE_ROUTE_ID have the constant value camelFailureRouteId. This allows for example you to fetch this information in your dead letter queue and use that for error reporting.
```

Control if Redelivery is Allowed During Stopping/Shutdown

Available as of Camel 2.11

Before **Camel 2.10**, Camel would perform redelivery while stopping a route, or shutting down Camel. This has improved a bit in **Camel 2.10**: Camel will no longer perform redelivery attempts when shutting down aggressively, e.g., during Graceful Shutdown and timeout hit.

From **Camel 2.11**: there is a new option `allowRedeliveryWhileStopping` which you can use to control if redelivery is allowed or not; notice that any in progress redelivery will still be executed. This option can only disallow any redelivery to be executed **after** the stopping of a route/shutdown of Camel has been triggered. If a redelivery is disallowed then a `RejectedExecutionException` is set on the Exchange and the processing of the Exchange stops. This means any consumer will see the Exchange as failed due the `RejectedExecutionException`. The default value is `true` for backward compatibility.

For example, the following snippet shows how to do this with Java DSL and XML DSL:

```
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/RedeliveryErrorHandlerNoRedeliveryOnShutdownTest.java}And the sample
sample with XML DSL{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringRedeliveryErrorHandlerNoRedeliveryOnShutdownTest.xml}
```

Samples

The following example shows how to configure the Dead Letter Channel configuration using the DSL{snippet:id=e3|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/builder/ErrorHandleTest.java}You can also configure the RedeliveryPolicy as this example shows{snippet:id=e4|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/builder/ErrorHandleTest.java}

How Can I Modify the Exchange Before Redelivery?

We support directly in Dead Letter Channel to set a Processor that is executed **before** each redelivery attempt. When Dead Letter Channel is doing redelivery its possible to configure a Processor that is executed just **before** every redelivery attempt. This can be used for the situations where you need to alter the message before its redelivered. Here we configure the Dead Letter Channel to use our processor `MyRedeliveryProcessor` to be executed before each redelivery.

```
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/DeadLetterChannelOnRedeliveryTest.java}And this is the processor
MyRedeliveryProcessor where we alter the message.{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/DeadLetterChannelOnRedeliveryTest.java}
```

How Can I Log What Caused the Dead Letter Channel to be Invoked?

You often need to know what went wrong that caused the Dead Letter Channel to be used and it does not offer logging for this purpose. So the Dead Letter Channel's endpoint can be set to a endpoint of our own (such as `direct:deadLetterChannel1`). We write a route to accept this Exchange and log the Exception, then forward on to where we want the failed Exchange moved to (which might be a DLQ queue for instance). See also <http://stackoverflow.com/questions/13711462/logging-camel-exceptions-and-sending-to-the-dead-letter-channel>

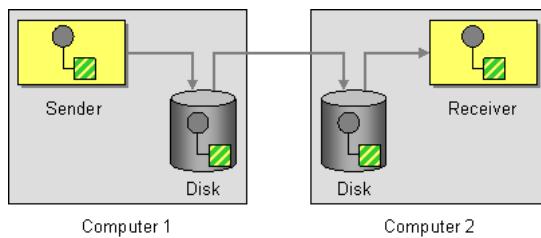
Using This Pattern

- Error Handler
- Exception Clause

Guaranteed Delivery

Camel supports the Guaranteed Delivery from the EIP patterns using among others 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, or use any of the many other database component such as SQL, JDBC, iBATIS/MyBatis, Hibernate
- HawtDB for a lightweight key-value persistent store



Example

The following example demonstrates illustrates the use of Guaranteed Delivery within the JMS component. By default, a message is not considered successfully delivered until the recipient has persisted the message locally guaranteeing its receipt in the event the destination becomes unavailable.

Using the Fluent Builders

```
from("direct:start")
    .to("jms:queue:foo");
```

Using the Spring XML Extensions

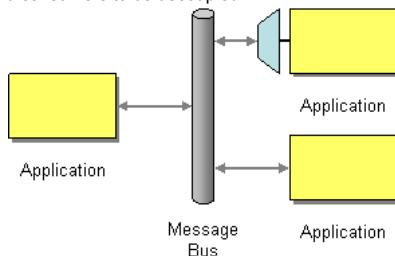
```
<route>
    <from uri="direct:start"/>
    <to uri="jms:queue:foo"/>
</route>
```

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 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 note is the XMPP component for supporting messaging over XMPP (Jabber)

Of course there are also ESB products such as Apache ServiceMix which serve as full fledged message busses. You can interact with Apache ServiceMix from Camel in many ways, but in particular you can use the NMR or JBI component to access the ServiceMix message bus directly.

Example

The following demonstrates how the Camel message bus can be used to communicate with consumers and producers

Using the Fluent Builders

```
from("direct:start")
    .pollEnrich("file:inbox?fileName=data.txt")
    .to("jms:queue:foo");
```

Using the Spring XML Extensions

```
<route>
    <from uri="direct:start"/>
    <pollEnrich uri="file:inbox?fileName=data.txt"/>
    <to uri="jms:queue:foo"/>
</route>
```

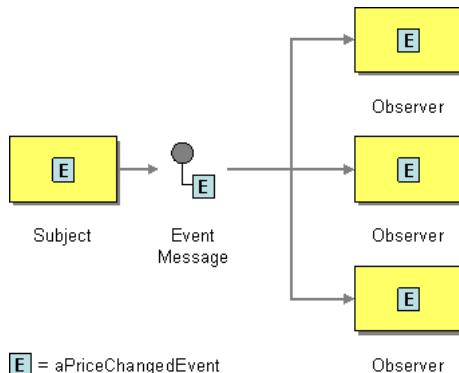
Using This Pattern

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 Construction

EVENT MESSAGE

Camel supports the Event Message from the EIP patterns by supporting the Exchange Pattern on a Message which can be set to **InOnly** to indicate a oneway event message. Camel Components then implement this pattern using the underlying transport or protocols.



The default behaviour of many Components is **InOnly** such as for JMS, File or SEDA Related

See the related Request Reply message.

Explicitly specifying InOnly

If you are using a component which defaults to **InOut** you can override the Exchange Pattern for an endpoint using the **pattern** property.

```
foo:bar?exchangePattern=InOnly
```

From 2.0 onwards on Camel you can specify the Exchange Pattern using the DSL.

Using the Fluent Builders

```
from("mq:someQueue").
    setExchangePattern(ExchangePattern.InOnly).
    bean(Foo.class);
```

or you can invoke an endpoint with an explicit pattern

```
from("mq:someQueue").
    inOnly("mq:anotherQueue");
```

Using the Spring XML Extensions

```
<route>
    <from uri="mq:someQueue"/>
    <inOnly uri="bean:foo"/>
</route>
```

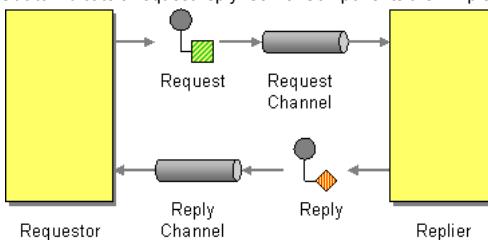
```
<route>
    <from uri="mq:someQueue"/>
    <inOnly uri="mq:anotherQueue"/>
</route>
```

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.

REQUEST REPLY

Camel supports the Request Reply from the EIP patterns by supporting the Exchange Pattern on a Message which can be set to InOut to indicate a request/reply. Camel Components then implement this pattern using the underlying transport or protocols.



For example when using JMS with InOut the component will by default perform these actions

- create by default a temporary inbound queue
- set the JMSReplyTo destination on the request message
- set the JMSCorrelationID on the request message
- send the request message
- consume the response and associate the inbound message to the request using the JMSCorrelationID (as you may be performing many concurrent request/responses).

Related

See the related Event Message message

Explicitly specifying InOut

When consuming messages from JMS a Request-Reply is indicated by the presence of the **JMSReplyTo** header.

You can explicitly force an endpoint to be in Request Reply mode by setting the exchange pattern on the URI. e.g.

```
jms:MyQueue?exchangePattern=InOut
```

You can specify the exchange pattern in DSL rule or Spring configuration.

```
// Send to an endpoint using InOut
from("direct:testInOut").inOut("mock:result");

// Send to an endpoint using InOut
from("direct:testInOnly").inOnly("mock:result");

// Set the exchange pattern to InOut, then send it from direct:inOnly to mock:result endpoint
from("direct:testSetToInOnlyThenTo")
    .setExchangePattern(ExchangePattern.InOnly)
    .to("mock:result");
from("direct:testSetInOutThenTo")
    .setExchangePattern(ExchangePattern.InOut)
```

```

.to("mock:result");

// Or we can pass the pattern as a parameter to the to() method
from("direct:testToInOnlyParam").to(ExchangePattern.InOnly, "mock:result");
from("direct:testToInOutParam").to(ExchangePattern.InOut, "mock:result");
from("direct:testToWithRobustInOnlyParam").to(ExchangePattern.RobustInOnly, "mock:result");

// Set the exchange pattern to InOut, then send it on
from("direct:testSetExchangePatternInOnly")
.setExchangePattern(ExchangePattern.InOnly).to("mock:result");

<camelContext xmlns="http://camel.apache.org/schema/spring">
<!-- Send the exchange as InOnly -->
<route>
<from uri="direct:testInOut"/>
<inOut uri="mock:result"/>
</route>

<!-- Send the exchange as InOnly -->
<route>
<from uri="direct:testInOnly"/>
<inOnly uri="mock:result"/>
</route>

<!-- lets set the exchange pattern then send it on -->
<route>
<from uri="direct:testSetToInOnlyThenTo"/>
<setExchangePattern pattern="InOnly"/>
<to uri="mock:result"/>
</route>
<route>
<from uri="direct:testSetToInOutThenTo"/>
<setExchangePattern pattern="InOut"/>
<to uri="mock:result"/>
</route>
<route>
<from uri="direct:testSetExchangePatternInOnly"/>
<setExchangePattern pattern="InOnly"/>
<to uri="mock:result"/>
</route>

<!-- Lets pass the pattern as an argument in the to element -->
<route>
<from uri="direct:testToInOnlyParam"/>
<to uri="mock:result" pattern="InOnly"/>
</route>
<route>
<from uri="direct:testToInOutParam"/>
<to uri="mock:result" pattern="InOut"/>
</route>
<route>
<from uri="direct:testToWithRobustInOnlyParam"/>
<to uri="mock:result" pattern="RobustInOnly"/>
</route>
</camelContext>

```

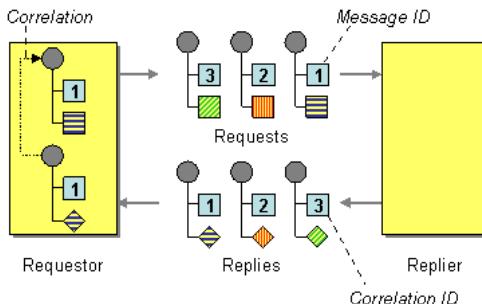
Using This Pattern

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.

Correlation Identifier

Camel supports the Correlation Identifier from the EIP patterns by getting or setting a header on a Message.

When working with the ActiveMQ or JMS components the correlation identifier header is called **JMSCorrelationID**. You can add your own correlation identifier to any message exchange to help correlate messages together to a single conversation (or business process).



The use of a Correlation Identifier is key to working with the Camel Business Activity Monitoring Framework and can also be highly useful when testing with simulation or canned data such as with the Mock testing framework

Some EIP patterns will spin off a sub message, and in those cases, Camel will add a correlation id on the Exchange as a property with key Exchange.CORRELATION_ID, which links back to the source Exchange. For example the Splitter, Multicast, Recipient List, and Wire Tap EIP does this.

The following example demonstrates using the Camel JMSMessageID as the Correlation Identifier within a request/reply pattern in the JMS component

Using the Fluent Builders

```
from("direct:start")
    .to(ExchangePattern.InOut, "jms:queue:foo?useMessageIDAsCorrelationID=true")
    .to("mock:result");
```

Using the Spring XML Extensions

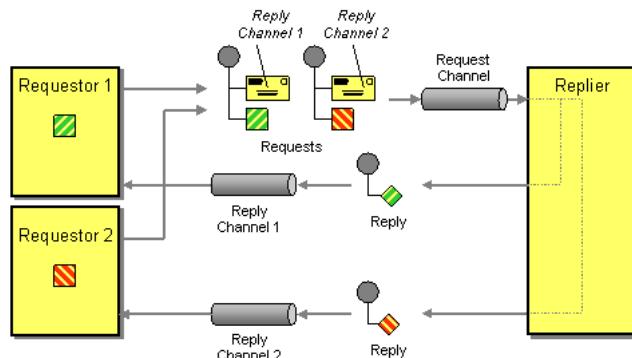
```
<route>
    <from uri="direct:start"/>
    <to uri="jms:queue:foo?useMessageIDAsCorrelationID=true" pattern="InOut"/>
    <to uri="mock:result"/>
</route>
```

See Also

- BAM

RETURN ADDRESS

Camel supports the Return Address from the EIP patterns by using the `JMSReplyTo` header.



For example when using JMS with InOut the component will by default return to the address given in `JMSReplyTo`.

Requestor Code

```
getMockEndpoint("mock:bar").expectedBodiesReceived("Bye World");
template.sendBodyAndHeader("direct:start", "World", "JMSReplyTo", "queue:bar");
```

Route Using the Fluent Builders

```
from("direct:start").to("activemq:queue:foo?preserveMessageQos=true");
from("activemq:queue:foo").transform(body().prepend("Bye "));
from("activemq:queue:bar?disableReplyTo=true").to("mock:bar");
```

Route Using the Spring XML Extensions

```
<route>
    <from uri="direct:start"/>
    <to uri="activemq:queue:foo?preserveMessageQos=true"/>
</route>

<route>
    <from uri="activemq:queue:foo"/>
    <transform>
        <simple>Bye ${in.body}</simple>
    </transform>
</route>

<route>
    <from uri="activemq:queue:bar?disableReplyTo=true"/>
    <to uri="mock:bar"/>
</route>
```

For a complete example of this pattern, see this junit test case

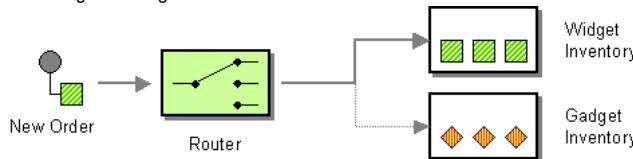
Using This Pattern

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 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 `seda:a` endpoint to either `seda:b`, `seda:c` or `seda:d` depending on the evaluation of various Predicate expressions

Using the Fluent Builders

```

RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("mock:error"));

        from("direct:a")
            .choice()
                .when(header("foo").isEqualTo("bar"))
                    .to("direct:b")
                .when(header("foo").isEqualTo("cheese"))
                    .to("direct:c")
                .otherwise()
                    .to("direct:d");
    }
};
  
```

See Why can I not use when or otherwise in a Java Camel route if you have problems with the Java DSL, accepting using when or otherwise.

Using the Spring XML Extensions

```

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

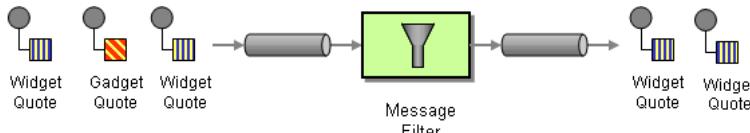
For further examples of this pattern in use you could look at the junit test case

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{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/builder/RouteBuilderTest.java}You can, of course, use many different Predicate languages such as XPath, XQuery, SQL or various Scripting Languages. Here is an XPath example{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/XPathFilterTest.java}Here's another example of using a bean to define the filter behavior:

```
java<from uri="direct:start"> .filter().method(MyBean.class, "isGoldCustomer").to("mock:result").end() .to("mock:end"); public static class MyBean { public boolean isGoldCustomer(@Header("level") String level) { return level.equals("gold"); } }
```

Using the Spring XML Extensions{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/xml/buildSimpleRouteWithHeaderPredicate.xml}You can also use a method call expression (to call a method on a bean) in the Message Filter, as shown below:

```
<bean id="myBean" class="com.foo.MyBean"/> <camelContext xmlns="http://camel.apache.org/schema/spring"> <route> <from uri="direct:a"/> <filter> <method ref="myBean" method="isGoldCustomer"/> <to uri="direct:b"/> </filter> </route> </camelContext>Filtered Endpoint Required Inside </filter> Tag
```

Ensure you put the endpoint you want to filter <to uri="seda:b"/> before the closing </filter> tag or the filter will not be applied. From **Camel 2.8**: omitting this will result in an error.

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

Using stop()

Stop is a bit different than a message filter as it will filter out all messages and end the route entirely (filter only applies to its child processor). Stop is convenient to use in a Content Based Router when you for example need to stop further processing in one of the predicates.

In the example below we do not want to route messages any further that has the word bye in the message body. Notice how we prevent this in the when() predicate by using the .stop().{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/RouteStopTest.java}

How To Determine If An Exchange Was Filtered

Available as of Camel 2.5

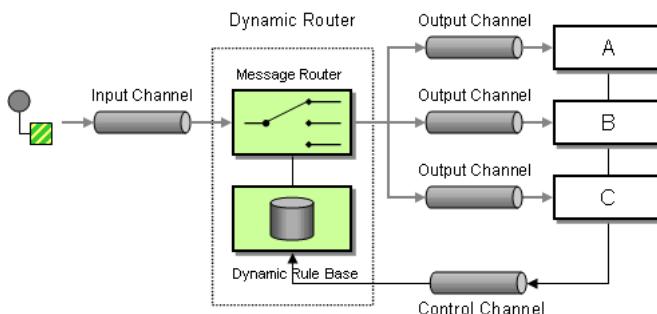
The Message Filter EIP will add a property on the Exchange that states if it was filtered or not.

The property has the key `Exchange.FILTER_MATCHED`, which has the String value of `camelFilterMatched`. Its value is a boolean indicating true or false. If the value is true then the Exchange was routed in the filter block. This property will be visible within the Message Filter block who's Predicate matches (value set to true), and to the steps immediately following the Message Filter with the value set based on the results of the last Message Filter Predicate evaluated.

Using This Pattern

DYNAMIC ROUTER

The Dynamic Router from the EIP patterns allows you to route messages while avoiding the dependency of the router on all possible destinations while maintaining its efficiency.



In Camel 2.5 we introduced a `dynamicRouter` in the DSL which is like a dynamic Routing Slip which evaluates the slip *on-the-fly*.

Beware

You must ensure the expression used for the `dynamicRouter` such as a bean, will return `null` to indicate the end. Otherwise the `dynamicRouter` will keep repeating endlessly.

Options

confluenceTableSmall

Name	Default Value	Description
<code>uriDelimiter</code>	,	Delimiter used if the Expression returned multiple endpoints.
<code>ignoreInvalidEndpoints</code>	false	If an endpoint uri could not be resolved, should it be ignored. Otherwise Camel will throw an exception stating the endpoint uri is not valid.
<code>cachesize</code>	1000	Camel 2.13.1/2.12.4: Allows to configure the cache size for the <code>ProducerCache</code> which caches producers for reuse in the routing slip. Will by default use the default cache size which is 1000. Setting the value to -1 allows to turn off the cache all together.

Dynamic Router in Camel 2.5 onwards

From Camel 2.5 the Dynamic Router will set a property (`Exchange.SLIP_ENDPOINT`) on the Exchange which contains the current endpoint as it advanced though the slip. This allows you to know how far we have processed in the slip. (It's a slip because the Dynamic Router implementation is based on top of Routing Slip).

Java DSL

In Java DSL you can use the `dynamicRouter` as shown below:

```
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/DynamicRouterTest.java}
```

Which will leverage a Bean to compute the slip *on-the-fly*, which could be implemented as follows:

```
{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/DynamicRouterTest.java}
```

Mind that this example is only for show and tell. The current implementation is not thread safe. You would have to store the state on the Exchange, to ensure thread safety, as shown below:

```
{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/DynamicRouterExchangePropertiesTest.java}
```

You could also store state as message headers, but they are not guaranteed to be preserved during routing, where as properties on the Exchange are. Although there was a bug in the method call expression, see the warning below.

Using beans to store state

Mind that in Camel 2.9.2 or older, when using a Bean the state is not propagated, so you will have to use a Processor instead. This is fixed in Camel 2.9.3 onwards.

Spring XML

The same example in Spring XML would be:

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringDynamicRouterTest.xml}
```

@DynamicRouter annotation

You can also use the `@DynamicRouter` annotation, for example the Camel 2.4 example below could be written as follows. The `route` method would then be invoked repeatedly as the message is processed dynamically. The idea is to return the next endpoint uri where to go. Return `null` to indicate the end. You can return multiple endpoints if you like, just as the Routing Slip, where each endpoint is separated by a delimiter.

```
javapublic class MyDynamicRouter { @Consume(uri = "activemq:foo") @DynamicRouter public String route(@XPath("customer/id") String customerId, @Header("Location") String location, Document body) { // query a database to find the best match of the endpoint based on the input parameters // return the next endpoint uri, where to go. Return null to indicate the end. } }
```

Dynamic Router in Camel 2.4 or older

The simplest way to implement this is to use the `RecipientList` Annotation on a Bean method to determine where to route the message.

```
javapublic class MyDynamicRouter { @Consume(uri = "activemq:foo") @RecipientList public List<String> route(@XPath("customer/id") String customerId, @Header("Location") String location, Document body) { // query a database to find the best match of the endpoint based on the input parameters ... } }
```

In the above we can use the Parameter Binding Annotations to bind different parts of the Message to method parameters or use an Expression such as using XPath or XQuery.

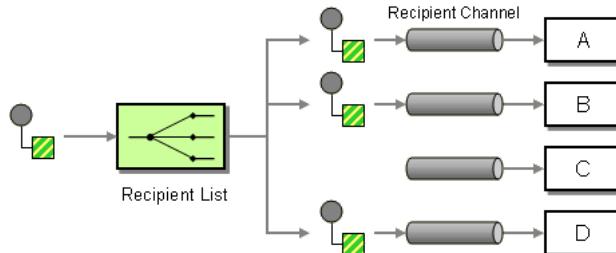
The method can be invoked in a number of ways as described in the Bean Integration such as

- POJO Producing
- Spring Remoting
- Bean component

Using This Pattern

Recipient List

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



The recipients will receive a copy of the **same Exchange**, and Camel will execute them sequentially.

Options

confluenceTableSmall

Name	Default Value	Description
delimiter	,	Delimiter used if the Expression returned multiple endpoints (like "direct:foo,direct:bar"). From Camel 2.13 onwards this can be disabled by setting delimiter to "false".
strategyRef		An AggregationStrategy that will assemble the replies from recipients into a single outgoing message from the Recipient List. By default Camel will use the last reply as the outgoing message. From Camel 2.12 onwards you can also use a POJO as the AggregationStrategy, see the Aggregator page for more details. If an exception is thrown from the aggregate method in the AggregationStrategy, then by default, that exception is not handled by the error handler. The error handler can be enabled to react if enabling the shareUnitOfWork option.
strategyMethodName		Camel 2.12: This option can be used to explicitly declare the method name to use, when using POJOs as the AggregationStrategy. See the Aggregator page for more details.
strategyMethodAllowNull	false	Camel 2.12: If this option is false then the aggregate method is not used if there was no data to enrich. If this option is true then null is used as the oldExchange (when no data to enrich), when using POJOs as the AggregationStrategy. See the Aggregator page for more details.
parallelProcessing	false	Camel 2.2: If enabled, messages are sent to the recipients concurrently. Note that the calling thread will still wait until all messages have been fully processed before it continues; it is the sending and processing of replies from recipients which happens in parallel.
parallelAggregate	false	Camel 2.14: If enabled then the aggregate method on AggregationStrategy can be called concurrently. Notice that this would require the implementation of AggregationStrategy to be implemented as thread-safe. By default this is false meaning that Camel synchronizes the call to the aggregate method. Though in some use-cases this can be used to archive higher performance when the AggregationStrategy is implemented as thread-safe.
executorServiceRef		Camel 2.2: A custom Thread Pool to use for parallel processing. Note that enabling this option implies parallel processing, so you need not enable that option as well.
stopOnException	false	Camel 2.2: Whether to immediately stop processing when an exception occurs. If disabled, Camel will send the message to all recipients regardless of any

		individual failures. You can process exceptions in an AggregationStrategy implementation, which supports full control of error handling.
ignoreInvalidEndpoints	false	Camel 2.3: Whether to ignore an endpoint URI that could not be resolved. If disabled, Camel will throw an exception identifying the invalid endpoint URI.
streaming	false	Camel 2.5: If enabled, Camel will process replies out-of-order - that is, in the order received in reply from each recipient. If disabled, Camel will process replies in the same order as specified by the Expression. So this specifies whether the response messages are aggregated as they come in, or in the exact order as the recipient list was evaluated. Only relevant if you enable parallelProcessing.
timeout		Camel 2.5: Specifies a processing timeout in milliseconds. If the Recipient List hasn't been able to send and process all replies within this timeframe, then the timeout triggers and the Recipient List breaks out, with message flow continuing to the next element. Note that if you provide a TimeoutAwareAggregationStrategy, its timeout method is invoked before breaking out. Beware: If the timeout is reached with running tasks still remaining, certain tasks (for which it is difficult for Camel to shut down in a graceful manner) may continue to run. So use this option with caution. We may be able to improve this functionality in future Camel releases.
onPrepareRef		Camel 2.8: A custom Processor to prepare the copy of the Exchange each recipient will receive. This allows you to perform arbitrary transformations, such as deep-cloning the message payload (or any other custom logic).
shareUnitofwork	false	Camel 2.8: Whether the unit of work should be shared. See the same option on Splitter for more details.
cachesize	1000	Camel 2.13.1/2.12.4: Allows to configure the cache size for the ProducerCache which caches producers for reuse in the recipient list. Will by default use the default cache size which is 1000. Setting the value to -1 allows to turn off the cache completely.

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 Annotations

You can use the RecipientList Annotation on a POJO to create a Dynamic Recipient List. For more details see the Bean Integration.

Using the Fluent Builders{snippet:id=multicast|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/builder/RouteBuilderTest.java}**Using the Spring XML Extensions**{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/xml/buildStaticRecipientList.xml}

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 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{snippet:id=e9|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/builder/RouteBuilderTest.java}The above assumes that the header contains a list of endpoint URIs. The following takes a single string header and tokenizes it{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/RecipientListTest.java}

Iterable value

The dynamic list of recipients that are defined in the header must be iterable such as:

- `java.util.collection`
- `java.util.Iterator`
- arrays
- `org.w3c.dom.NodeList`
- a single String with values separated by comma
- any other type will be regarded as a single value

Using the Spring XML Extensions{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/xml/buildDynamicRecipientList.xml}For further examples of this pattern in action you could take a look at one of the junit test cases.

Using delimiter in Spring XML

In Spring DSL you can set the `delimiter` attribute for setting a delimiter to be used if the header value is a single String with multiple separated endpoints. By default Camel uses comma as delimiter, but this option lets you specify a custom delimiter to use instead.`{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/recipientListWithStringDelimitedHeader.xml}` So if `myHeader` contains a String with the value "activemq:queue:foo, activemq:topic:hello, log:bar" then Camel will split the String using the delimiter given in the XML that was comma, resulting into 3 endpoints to send to. You can use spaces between the endpoints as Camel will trim the value when it lookup the endpoint to send to.

Note: In Java DSL you use the `tokenizer` to achieve the same. The route above in Java DSL:
`from("direct:a").recipientList(header("myHeader")).tokenize(",");`

In **Camel 2.1** its a bit easier as you can pass in the delimiter as 2nd parameter:
`from("direct:a").recipientList(header("myHeader"), "#");`

Sending to multiple recipients in parallel

Available as of Camel 2.2

The Recipient List now supports `parallelProcessing` that for example Splitter also supports. You can use it to use a thread pool to have concurrent tasks sending the Exchange to multiple recipients concurrently.

`from("direct:a").recipientList(header("myHeader")).parallelProcessing();`

And in Spring XML it is an attribute on the recipient list tag.

`<route> <from uri="direct:a"/> <recipientList parallelProcessing="true"> <header>myHeader</header> </recipientList> </route>`

Stop continuing in case one recipient failed

Available as of Camel 2.2

The Recipient List now supports `stopOnException` that for example Splitter also supports. You can use it to stop sending to any further recipients in case any recipient failed.

`from("direct:a").recipientList(header("myHeader")).stopOnException();`

And in Spring XML its an attribute on the recipient list tag.

`<route> <from uri="direct:a"/> <recipientList stopOnException="true"> <header>myHeader</header> </recipientList> </route>`

Note: You can combine `parallelProcessing` and `stopOnException` and have them both true.

Ignore invalid endpoints

Available as of Camel 2.3

The Recipient List now supports `ignoreInvalidEndpoints` (like the Routing Slip). You can use it to skip endpoints which are invalid.

`from("direct:a").recipientList(header("myHeader")).ignoreInvalidEndpoints();`

And in Spring XML it is an attribute on the recipient list tag.

`<route> <from uri="direct:a"/> <recipientList ignoreInvalidEndpoints="true"> <header>myHeader</header> </recipientList> </route>`

Then let us say the `myHeader` contains the following two endpoints `direct:foo,xxx:bar`. The first endpoint is valid and works. However the second one is invalid and will just be ignored. Camel logs at INFO level about it, so you can see why the endpoint was invalid.

Using custom AggregationStrategy

Available as of Camel 2.2

You can now use your own AggregationStrategy with the Recipient List. However this is rarely needed. What it is good for is that in case you are using Request Reply messaging then the replies from the recipients can be aggregated. By default Camel uses `useLatestAggregationStrategy` which just keeps that last received reply. If you must remember all the bodies that all the recipients sent back, then you can use your own custom aggregator that keeps those. It is the same principle as with the Aggregator EIP so check it out for details.

`from("direct:a").recipientList(header("myHeader")).aggregationStrategy(new MyOwnAggregationStrategy()).to("direct:b");`

And in Spring XML it is again an attribute on the recipient list tag.

`<route> <from uri="direct:a"/> <recipientList strategyRef="myStrategy"> <header>myHeader</header> </recipientList> <to uri="direct:b"/> </route> <bean id="myStrategy" class="com.mycompany.MyOwnAggregationStrategy"/>`

Knowing which endpoint when using custom AggregationStrategy

Available as of Camel 2.12

When using a custom AggregationStrategy then the aggregate method is always invoked in sequential order (also if parallel processing is enabled) of the endpoints the Recipient List is using. However from Camel 2.12 onwards this is easier to know as the `newExchange` Exchange now has a property stored (key is `Exchange.RECIPIENT_LIST_ENDPOINT`) with the uri of the Endpoint. So you know which endpoint you are aggregating from. The code block shows how to access this property in your Aggregator.

```
@Override public Exchange aggregate(Exchange oldExchange, Exchange newExchange) { String uri = newExchange.getProperty(Exchange.RECIPIENT_LIST_ENDPOINT, String.class); ... }
```

Using custom thread pool

Available as of Camel 2.2

A thread pool is only used for parallelProcessing. You supply your own custom thread pool via the ExecutorServiceStrategy (see Camel's Threading Model), the same way you would do it for the aggregationStrategy. By default Camel uses a thread pool with 10 threads (subject to change in future versions).

Using method call as recipient list

You can use a Bean to provide the recipients, for example:

```
from("activemq:queue:test").recipientList().method(MessageRouter.class, "routeTo");
```

And then MessageRouter:

```
public class MessageRouter { public String routeTo() { String queueName = "activemq:queue:test2"; return queueName; } }
```

When you use a Bean then do **not** use the @RecipientList annotation as this will in fact add yet another recipient list, so you end up having two. Do **not** do the following.

```
public class MessageRouter { @RecipientList public String routeTo() { String queueName = "activemq:queue:test2"; return queueName; } }
```

You should only use the snippet above (using @RecipientList) if you just route to a Bean which you then want to act as a recipient list.

So the original route can be changed to:

```
from("activemq:queue:test").bean(MessageRouter.class, "routeTo");
```

Which then would invoke the routeTo method and detect that it is annotated with @RecipientList and then act accordingly as if it was a recipient list EIP.

Using timeout

Available as of Camel 2.5

If you use parallelProcessing then you can configure a total timeout value in millis. Camel will then process the messages in parallel until the timeout is hit. This allows you to continue processing if one message consumer is slow. For example you can set a timeout value of 20 sec.

Tasks may keep running

If the timeout is reached with running tasks still remaining, certain tasks for which it is difficult for Camel to shut down in a graceful manner may continue to run. So use this option with a bit of care. We may be able to improve this functionality in future Camel releases.

For example in the unit test below you can see that we multicast the message to 3 destinations. We have a timeout of 2 seconds, which means only the last two messages can be completed within the timeframe. This means we will only aggregate the last two which yields a result aggregation which outputs "BC".
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/MulticastParallelTimeoutTest.java}

Timeout in other EIPs

This timeout feature is also supported by Splitter and both multicast and recipientList.

By default if a timeout occurs the AggregationStrategy is not invoked. However you can implement a special version javaTimeoutAwareAggregationStrategyTimeoutAwareAggregationStrategypublic interface TimeoutAwareAggregationStrategy extends AggregationStrategy { *** A timeout occurred ** @param oldExchange the oldest exchange (is <tt>null</tt> on first aggregation as we only have the new exchange) * @param index the index * @param total the total * @param timeout the timeout value in millis */ void timeout(Exchange oldExchange, int index, int total, long timeout);

This allows you to deal with the timeout in the AggregationStrategy if you really need to.

Timeout is total

The timeout is total, which means that after X time, Camel will aggregate the messages which have completed within the timeframe. The remainders will be cancelled. Camel will also only invoke the timeout method in the TimeoutAwareAggregationStrategy once, for the first index which caused the timeout.

Using onPrepare to execute custom logic when preparing messages

Available as of Camel 2.8

See details at Multicast

Using ExchangePattern in recipients

Available as of Camel 2.15

The recipient list will by default use the current Exchange Pattern. Though one can imagine use-cases where one wants to send a message to a recipient using a different exchange pattern. For example you may have a route that initiates as an InOnly route, but want to use InOut exchange pattern with a recipient list. To do this in earlier Camel releases, you would need to change the exchange pattern before the recipient list, or use onPrepare option to alter the pattern. From Camel 2.15 onwards, you can configure the exchange pattern directly in the recipient endpoints.

For example in the route below we pick up new files (which will be started as InOnly) and then route to a recipient list. As we want to use InOut with the ActiveMQ (JMS) endpoint we can now specify this using the exchangePattern=InOut option. Then the response from the JMS request/reply will then be continued routed, and thus the response is what will be stored in as a file in the outbox directory.

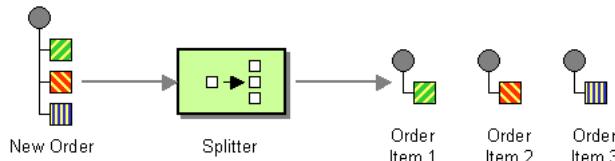
```
from("file:inbox") // the exchange pattern is InOnly initially when using a file route .recipientList().constant("activemq:queue:inbox");
exchangePattern=InOut").to("file:outbox");
The recipient list will not alter the original exchange pattern. So in the example above the exchange pattern will still be InOnly
when the message is routed to the file:outbox endpoint.
```

If you want to alter the exchange pattern permanently then use the `.setExchangePattern` option. See more details at Request Reply and Event Message.

Using This Pattern

Splitter

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



You need to specify a Splitter as `split()`. In earlier versions of Camel, you need to use `splitter()`.

Options

confluenceTableSmall

Name	Default Value	Description
strategyRef		Refers to an AggregationStrategy to be used to assemble the replies from the sub-messages, into a single outgoing message from the Splitter. See the defaults described below in <i>What the Splitter returns</i> . From Camel 2.12 onwards you can also use a POJO as the AggregationStrategy, see the Aggregate page for more details. If an exception is thrown from the aggregate method in the AggregationStrategy, then by default, that exception is not handled by the error handler. The error handler can be enabled to react if enabling the <code>shareUnitOfWork</code> option.
strategyMethodName		Camel 2.12: This option can be used to explicit declare the method name to use, when using POJOs as the AggregationStrategy. See the Aggregate page for more details.
strategyMethodAllowNull	false	Camel 2.12: If this option is false then the aggregate method is not used for the very first splitted message. If this option is true then null values is used as the oldexchange (for the very first message splitted), when using POJOs as the AggregationStrategy. See the Aggregate page for more details.
parallelProcessing	false	If enabled then processing the sub-messages occurs concurrently. Note the caller thread will still wait until all sub-messages has been fully processed, before it continues.
parallelAggregate	false	Camel 2.14: If enabled then the aggregate method on AggregationStrategy can be called concurrently. Notice that this would require the implementation of AggregationStrategy to be implemented as thread-safe. By default this is false meaning that Camel synchronizes the call to the aggregate method. Though in some use-cases this can be used to achieve higher performance when the AggregationStrategy is implemented as thread-safe.
executorServiceRef		Refers to a custom Thread Pool to be used for parallel processing. Notice if you set this option, then parallel processing is automatically implied, and you do not have to enable that option as well.
stopOnException	false	Camel 2.2: Whether or not to stop continue processing immediately when an exception occurred. If disable, then Camel continue splitting and process the sub-messages regardless if one of them failed. You can deal with exceptions in the AggregationStrategy class where you have full control how to handle that.
streaming	false	If enabled then Camel will split in a streaming fashion, which means it will split the input message in chunks. This reduces the memory overhead. For example if you split big messages its recommended to enable streaming. If streaming is enabled then the sub-message replies will be aggregated out-of-order, eg in the order they come back. If disabled, Camel will process sub-message replies in the same order as they were splitted.
timeout		

Camel 2.5: Sets a total timeout specified in millis. If the Recipient List hasn't been able to split and process all replies within the given timeframe, then the timeout triggers and the Splitter breaks out and continues. Notice if you provide a TimeoutAwareAggregationStrategy then the `timeout` method is invoked before breaking out. If the timeout is reached with running tasks still remaining, certain tasks for which it is difficult for Camel to shut down in a graceful manner may continue to run. So use this option with a bit of care. We may be able to improve this functionality in future Camel releases.

onPrepareRef		Camel 2.8: Refers to a custom Processor to prepare the sub-message of the Exchange, before its processed. This allows you to do any custom logic, such as deep-cloning the message payload if that's needed etc.
shareunitofwork	false	Camel 2.8: Whether the unit of work should be shared. See further below for more details.

Exchange properties

The following properties are set on each Exchange that are split:

property	type	description
CamelSplitIndex	int	A split counter that increases for each Exchange being split. The counter starts from 0.
CamelSplitsize	int	The total number of Exchanges that was splitted. This header is not applied for stream based splitting. From Camel 2.9 onwards this header is also set in stream based splitting, but only on the completed Exchange.
CamelsplitComplete	boolean	Camel 2.4: Whether or not this Exchange is the last.

Examples

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

Using the Fluent Builders
`{snippet:id=splitter|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/builder/RouteBuilderTest.java}` 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").split(xpath("//foo/bar")).convertBodyTo(String.class).to("file://some/directory")`
Using the Spring XML Extensions
`{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/xml/buildSplitter.xml}` For further examples of this pattern in use you could look at one of the junit test case

Splitting a Collection, Iterator or Array

A common use case is to split a Collection, Iterator or Array from the message. In the sample below we simply use an Expression to identify the value to split.

```
javafrom("direct:splitUsingBody").split(body()).to("mock:result");
from("direct:splitUsingHeader").split(header("foo")).to("mock:result");
In Spring XML you can use the Simple language to identify the value to split.
xml<split> <simple>${body}</simple> <to uri="mock:result"/> </split> <split> <simple>${header.foo}</simple> <to
uri="mock:result"/> </split>
```

Using Tokenizer from Spring XML Extensions*

You can use the tokenizer expression in the Spring DSL to split bodies or headers using a token. This is a common use-case, so we provided a special `tokenizer` tag for this.

In the sample below we split the body using a `@` as separator. You can of course use comma or space or even a regex pattern, also set `regex=true`.
`{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/splitterTokenizerTest.xml}`

What the Splitter returns

Camel 2.2 or older:

The Splitter will by default return the **last** splitted message.

Camel 2.3 and newer

The Splitter will by default return the original input message.

For all versions

You can override this by supplying your own strategy as an AggregationStrategy. There is a sample on this page (Split aggregate request/reply sample). Notice its the same strategy as the Aggregator supports. This Splitter can be viewed as having a build in light weight Aggregator.

Parallel execution of distinct 'parts'

If you want to execute all parts in parallel you can use special notation of `split()` 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").split(xPathBuilder, true).to("activemq:my.parts");
```

The boolean option has been refactored into a builder method `parallelProcessing` so its easier to understand what the route does when we use a method instead of `true|false`.

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

Stream based

Splitting big XML payloads

The XPath engine in Java and saxon will load the entire XML content into memory. And thus they are not well suited for very big XML payloads.

Instead you can use a custom Expression which will iterate the XML payload in a streamed fashion. From Camel 2.9 onwards you can use the Tokenizer language

which supports this when you supply the start and end tokens. From Camel 2.14, you can use the XMLTokenizer language which is specifically provided for tokenizing XML documents.

You can split streams by enabling the streaming mode using the `streaming` builder method.

```
from("direct:streaming").split(body().tokenize(",")).streaming().to("activemq:my.parts");
```

You can also supply your custom splitter to use with streaming like this:

```
import static org.apache.camel.builder.ExpressionBuilder.beanExpression; from("direct:streaming") .split(beanExpression(new MyCustomIteratorFactory(), "iterator")) .streaming().to("activemq:my.parts")
```

Streaming big XML payloads using Tokenizer language

There are two tokenizers that can be used to tokenize an XML payload. The first tokenizer uses the same principle as in the text tokenizer to scan the XML payload and extract a sequence of tokens.

Available as of Camel 2.9

If you have a big XML payload, from a file source, and want to split it in streaming mode, then you can use the Tokenizer language with start/end tokens to do this with low memory footprint.

StAX component

The Camel StAX component can also be used to split big XML files in a streaming mode. See more details at StAX.

For example you may have a XML payload structured as follows

```
xml<orders> <order> <!-- order stuff here --> </order> <order> <!-- order stuff here --> </order> ... <order> <!-- order stuff here -->
</order> </orders>
```

Now to split this big file using XPath would cause the entire content to be loaded into memory. So instead we can use the Tokenizer language to do this as follows:

```
from("file:inbox") .split().tokenizeXML("order").streaming() .to("activemq:queue:order");
```

In XML DSL the route would be as follows:

```
xml<route> <from uri="file:inbox"/> <split streaming="true"> <tokenize token="order" xml="true"/> <to
uri="activemq:queue:order"/> </split> </route>
```

Notice the `tokenizeXML` method which will split the file using the tag name of the child node (more precisely speaking, the local name of the element without its namespace prefix if any), which mean it will grab the content between the `<order>` and `</order>` tags (incl. the tokens). So for example a splitted message would be as follows:

```
xml <order> <!-- order stuff here --> </order>
```

If you want to inherit namespaces from a root/parent tag, then you can do this as well by providing the name of the root/parent tag:

```
xml<route> <from uri="file:inbox"/> <split streaming="true"> <tokenize token="order" inheritNamespaceTagName="orders"
xml="true"/> <to uri="activemq:queue:order"/> </split> </route>
```

And in Java DSL its as follows:

```
from("file:inbox") .split().tokenizeXML("order", "orders").streaming() .to("activemq:queue:order");
```

Available as of Camel 2.13.1, you can set the above `inheritNamespaceTagName` property to `**` to include the preceding context in each token (i.e., generating each token enclosed in its ancestor elements). It is noted that each token must share the same ancestor elements in this case.

The above tokenizer works well on simple structures but has some inherent limitations in handling more complex XML structures.

Available as of Camel 2.14

The second tokenizer uses a StAX parser to overcome these limitations. This tokenizer recognizes XML namespaces and also handles simple and complex XML structures more naturally and efficiently.

To split using this tokenizer at `{urn:shop}order`, we can write

```
Namespaces ns = new Namespaces("ns1", "urn:shop"); ... from("file:inbox") .split().xtokenize("//ns1:order", 'i', ns).streaming()
.to("activemq:queue:order")
```

Two arguments control the behavior of the tokenizer. The first argument specifies the element using a path notation. This path notation uses a subset of xpath with wildcard support. The second argument represents the extraction mode. The available

extraction modes are:

mode	description
i	injecting the contextual namespace bindings into the extracted token (default)
w	wrapping the extracted token in its ancestor context
u	unwrapping the extracted token to its child content
t	extracting the text content of the specified element

Having an input XML

```
xml<m:orders xmlns:m="urn:shop" xmlns:cat="urn:shop:catalog"> <m:order><id>123</id><date>2014-02-25</date>...</m:order>
```

...

Each mode will result in the following tokens,

i	<m:order xmlns:m="urn:shop" xmlns:cat="urn:shop:catalog"><id>123</id><date>2014-02-25</date>...
w	<m:orders xmlns:m="urn:shop" xmlns:cat="urn:shop:catalog"> <m:order><id>123</id><date>2014-02-25</date>...</m:order></m:orders>
u	<id>123</id><date>2014-02-25</date>...
t	1232014-02-25...

In XML DSL, the equivalent route would be written as follows:

```
xml<camelContext xmlns:ns1="urn:shop"> <route> <from uri="file:inbox"/> <split streaming="true">
```

```
<xtokenize>/ns1:order</xtokenize> <to uri="activemq:queue:order"/> </split> </route> </camelContext>
or setting the extraction mode explicitly as
xml ...<xtokenize mode="i">/>/ns1:order</xtokenize> ...
```

Note that this StAX based tokenizer's uses StAX Location API and requires a StAX Reader implementation (e.g., woodstox) that correctly returns the offset position pointing to the beginning of each event triggering segment (e.g., the offset position of '<' at each start and end element event). If you use a StAX Reader which does not implement that API correctly it results in invalid xml snippets after the split. For example the snippet could be wrong terminated:

```
<Start>...</Start> .... <Start>...</Start>
```

Splitting files by grouping N lines together

Available as of Camel 2.10

The Tokenizer language has a new option group that allows you to group N parts together, for example to split big files into chunks of 1000 lines.

```
from("file:inbox") .split().tokenize("\n", 1000).streaming() .to("activemq:queue:order");
And in XML DSL
```

```
xml<route> <from uri="file:inbox"/> <split streaming="true"> <tokenize token="\n" group="1000"/> <to
uri="activemq:queue:order"/> </split> </route>
```

The group option is a number that must be a positive number that dictates how many groups to combine together. Each part will be combined using the token.

So in the example above the message being sent to the activemq order queue, will contain 1000 lines, and each line separated by the token (which is a new line token).

The output when using the group option is always a `java.lang.String` type.

Specifying a custom aggregation strategy

This is specified similar to the Aggregator.

Specifying a custom ThreadPoolExecutor

You can customize the underlying ThreadPoolExecutor used in the parallel splitter. In the Java DSL try something like this:
`JavaXPathBuilder xPathBuilder = new XPathBuilder("//foo/bar"); ExecutorService pool = ... from("activemq:my.queue")
.split(xPathBuilder).executorService(pool) .to("activemq:my.parts");`

Using a Pojo to do the splitting

As the Splitter can use any Expression to do the actual splitting we leverage this fact and use a **method** expression to invoke a Bean to get the splitted parts.

The Bean should return a value that is iterable such as: `java.util.Collection`, `java.util.Iterator` or an array.
So the returned value, will then be used by Camel at runtime, to split the message.

Streaming mode and using pojo

When you have enabled the streaming mode, then you should return a `Iterator` to ensure streamish fashion. For example if the message is a big file, then by using an iterator, that returns a piece of the file in chunks, in the next method of the `Iterator`

ensures low memory footprint. This avoids the need for reading the entire content into memory. For an example see the source code for the TokenizePair implementation.

In the route we define the Expression as a method call to invoke our Bean that we have registered with the id mySplitterBean in the Registry.{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/SplitterPojoTest.java}And the logic for our Bean is as simple as. Notice we use Camel Bean Binding to pass in the message body as a String object.{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/SplitterPojoTest.java}

Split aggregate request/reply sample

This sample shows how you can split an Exchange, process each splitted message, aggregate and return a combined response to the original caller using request/reply.

The route below illustrates this and how the split supports a **aggregationStrategy** to hold the in progress processed messages:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/SplitAggregateInOutTest.java}And the OrderService bean is as follows:{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/SplitAggregateInOutTest.java}And our custom **aggregationStrategy** that is responsible for holding the in progress aggregated message that after the splitter is ended will be sent to the **buildCombinedResponse** method for final processing before the combined response can be returned to the waiting caller.{snippet:id=e3|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/SplitAggregateInOutTest.java}So lets run the sample and see how it works. We send an Exchange to the **direct:start** endpoint containing a IN body with the String value: A@B@C. The flow is: HandleOrder: A HandleOrder: B Aggregate old orders: (id=1,item=A) Aggregate new order: (id=2,item=B) HandleOrder: C Aggregate old orders: (id=1,item=A);(id=2,item=B) Aggregate new order: (id=3,item=C) BuildCombinedResponse: (id=1,item=A);(id=2,item=B);(id=3,item=C) Response to caller: Response[(id=1,item=A);(id=2,item=B);(id=3,item=C)]

Stop processing in case of exception

Available as of Camel 2.1

The Splitter will by default continue to process the entire Exchange even in case of one of the splitted message will throw an exception during routing.

For example if you have an Exchange with 1000 rows that you split and route each sub message. During processing of these sub messages an exception is thrown at the 17th. What Camel does by default is to process the remainder 983 messages. You have the chance to remedy or handle this in the **AggregationStrategy**.

But sometimes you just want Camel to stop and let the exception be propagated back, and let the Camel error handler handle it. You can do this in Camel 2.1 by specifying that it should stop in case of an exception occurred. This is done by the **stopOnException** option as shown below:

```
from("direct:start").split(body()).tokenize(",").stopOnException().process(new MyProcessor()).to("mock:split");
And using XML DSL you specify it as follows:
xml <route> <from uri="direct:start"/> <split stopOnException="true"> <tokenize token=","/> <process ref="myProcessor"/> <to uri="mock:split"/> </split> </route>
```

Using onPrepare to execute custom logic when preparing messages

Available as of Camel 2.8

See details at Multicast

Sharing unit of work

Available as of Camel 2.8

The Splitter will by default not share unit of work between the parent exchange and each splitted exchange. This means each sub exchange has its own individual unit of work.

For example you may have an use case, where you want to split a big message. And you want to regard that process as an atomic isolated operation that either is a success or failure. In case of a failure you want that big message to be moved into a dead letter queue. To support this use case, you would have to share the unit of work on the Splitter.

Here is an example in Java DSL:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/SplitSubUnitOfWorkTest.java}Now in this example what would happen is that in case there is a problem processing each sub message, the error handler will kick in (yes error handling still applies for the sub messages). **But** what doesn't happen is that if a sub message fails all redelivery attempts (its exhausted), then its **not** moved into that dead letter queue. The reason is that we have shared the unit of work, so the sub message will report the error on the shared unit of work. When the Splitter is done, it checks the state of the shared unit of work and checks if any errors occurred. And if an error occurred it will set the exception on the Exchange and mark it for rollback. The error handler will yet again kick in, as the Exchange has been marked as rollback and it had an exception as well. No redelivery attempts is performed (as it was marked for rollback) and the Exchange will be moved into the dead letter queue.

Using this from XML DSL is just as easy as you just have to set the **shareUnitOfWork** attribute to true:

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringSplitSubUnitOfWorkTest.xml}
```

Implementation of shared unit of work

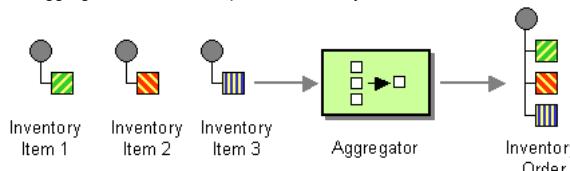
So in reality the unit of work is not shared as a single object instance. Instead subunitofwork is attached to their parent, and issues callback to the parent about their status (commit or rollback). This may be refactored in Camel 3.0 where larger API changes can be done.

Using This Pattern

Aggregator

This applies for Camel version 2.3 or newer. If you use an older version then use this Aggregator link instead.

The Aggregator from the EIP patterns allows you to combine a number of messages together into a single message.



A correlation Expression is used to determine the messages which should be aggregated together. If you want to aggregate all messages into a single message, just use a constant expression. An AggregationStrategy is used to combine all the message exchanges for a single correlation key into a single message exchange.

Aggregator options

The aggregator supports the following options:

[confluenceTableSmall](#)

Option	Default	Description
correlationExpression		Mandatory Expression which evaluates the correlation key to use for aggregation. The Exchange which has the same correlation key is aggregated together. If the correlation key could not be evaluated an Exception is thrown. You can disable this by using the ignorebadCorrelationkeys option.
aggregationStrategy		Mandatory AggregationStrategy which is used to merge the incoming Exchange with the existing already merged exchanges. At first call the oldExchange parameter is null. On subsequent invocations the oldExchange contains the merged exchanges and newExchange is of course the new incoming Exchange. From Camel 2.9.2 onwards the strategy can also be a TimeoutAwareAggregationStrategy implementation, supporting the timeout callback, see further below for more details. From Camel 2.16: the strategy can also be a PreCompletionAwareAggregationStrategy implementation which then runs the completion check in pre-completion mode. See further below for more details.
strategyRef		A reference to lookup the AggregationStrategy in the Registry. From Camel 2.12 onwards you can also use a POJO as the AggregationStrategy, see further below for details.
strategyMethodName		Camel 2.12: This option can be used to explicit declare the method name to use, when using POJOs as the AggregationStrategy. See further below for more details.
strategyMethodAllowNull	false	Camel 2.12: If this option is false then the aggregate method is not used for the very first aggregation. If this option is true then null values is used as the oldExchange (at the very first aggregation), when using POJOs as the AggregationStrategy. See further below for more details.
completionSize		Number of messages aggregated before the aggregation is complete. This option can be set as either a fixed value or using an Expression which allows you to evaluate a size dynamically - will use Integer as result. If both are set Camel will fallback to use the fixed value if the Expression result was null or 0.
completionTimeout		Time in millis that an aggregated exchange should be inactive before its complete. This option can be set as either a fixed value or using an Expression which allows you to evaluate a timeout dynamically - will use Long as result. If both are set Camel will fallback to use the fixed value if the Expression result was null or 0. You cannot use this option together with completionInterval, only one of the two can be used.
completionInterval		

		A repeating period in millis by which the aggregator will complete all current aggregated exchanges. Camel has a background task which is triggered every period. You cannot use this option together with completionTimeout, only one of them can be used.
completionPredicate		A Predicate to indicate when an aggregated exchange is complete. From Camel 2.15: if this is not specified and the AggregationStrategy object implements Predicate, the aggregationStrategy object will be used as the completionPredicate.
completionFromBatchConsumer	false	This option is if the exchanges are coming from a Batch Consumer. Then when enabled the Aggregator2 will use the batch size determined by the Batch Consumer in the message header camelBatchsize. See more details at Batch Consumer. This can be used to aggregate all files consumed from a File endpoint in that given poll.
forceCompletiononstop	false	Camel 2.9 Indicates to complete all current aggregated exchanges when the context is stopped
completeAllOnStop	false	Camel 2.16: Indicates to wait to complete all current and partial (pending) aggregated exchanges when the context is stopped. This also means that we will wait for all pending exchanges which are stored in the aggregation repository to complete so the repository is empty before we can stop. You may want to enable this when using the memory based aggregation repository that is memory based only, and do not store data on disk. When this option is enabled, then the aggregator is waiting to complete all those exchanges before its stopped, when stopping CamelContext or the route using it.
eagerCheckCompletion	false	Whether or not to eager check for completion when a new incoming Exchange has been received. This option influences the behavior of the completionPredicate option as the Exchange being passed in changes accordingly. When false the Exchange passed in the Predicate is the <i>aggregated</i> Exchange which means any information you may store on the aggregated Exchange from the AggregationStrategy is available for the Predicate. When true the Exchange passed in the Predicate is the <i>incoming</i> Exchange, which means you can access data from the incoming Exchange.
groupExchanges	false	If enabled then Camel will group all aggregated Exchanges into a single combined org.apache.camel.impl.GroupedExchange holder class that holds all the aggregated Exchanges. And as a result only one Exchange is being sent out from the aggregator. Can be used to combine many incoming Exchanges into a single output Exchange without coding a custom AggregationStrategy yourself. Note: this option does not support persistent repository with the aggregator. See further below for an example and more details.
ignoreInvalidCorrelationKeys	false	Whether or not to ignore correlation keys which could not be evaluated to a value. By default Camel will throw an Exception, but you can enable this option and ignore the situation instead.
closeCorrelationKeyOnCompletion		Whether or not too <i>late</i> Exchanges should be accepted or not. You can enable this to indicate that if a correlation key has already been completed, then any new exchanges with the same correlation key be denied. Camel will then throw a closedCorrelationKeyException exception. When using this option you pass in a integer which is a number for a LRUCache which keeps that last X number of closed correlation keys. You can pass in 0 or a negative value to indicate an unbounded cache. By passing in a number you are ensured that cache won't grow too big if you use a lot of different correlation keys.
discardOnCompletionTimeout	false	Camel 2.5: Whether or not exchanges which complete due to a timeout should be discarded. If enabled then when a timeout occurs the aggregated message will not be sent out but dropped (discarded).
aggregationRepository		Allows you to plugin your own implementation of org.apache.camel.spi.AggregationRepository which keeps track of the current inflight aggregated exchanges. Camel uses by default a memory based implementation.
aggregationRepositoryRef		Reference to lookup a aggregationRepository in the Registry.
parallelProcessing	false	When aggregated are completed they are being send out of the aggregator. This option indicates whether or not Camel should use a thread pool with multiple threads for concurrency. If no custom thread pool has been specified then Camel creates a default pool with 10 concurrent threads.

executorService		If using parallelProcessing you can specify a custom thread pool to be used. In fact also if you are not using parallelProcessing this custom thread pool is used to send out aggregated exchanges as well.
executorServiceRef		Reference to lookup a executorService in the Registry
timeoutCheckerExecutorService		Camel 2.9: If using either of the completionTimeout, completionTimeoutExpression, or completionInterval options a background thread is created to check for the completion for every aggregator. Set this option to provide a custom thread pool to be used rather than creating a new thread for every aggregator.
timeoutCheckerExecutorServiceRef		Camel 2.9: Reference to lookup a timeoutCheckerExecutorService in the Registry
optimisticLocking	false	Camel 2.11: Turns on using optimistic locking, which requires the aggregationRepository being used, is supporting this by implementing the org.apache.camel.spi.OptimisticLockingAggregationRepository interface.
optimisticLockRetryPolicy		Camel 2.11.1: Allows to configure retry settings when using optimistic locking.

Exchange Properties

The following properties are set on each aggregated Exchange:

confluenceTableSmall

Header	Type	Description
CamelAggregatedSize	int	The total number of Exchanges aggregated into this combined Exchange.
CamelAggregatedCompletedBy	String	Indicator how the aggregation was completed as a value of either: predicate, size_strategy, consumer, timeout, forceCompletion or interval.

About AggregationStrategy

The AggregationStrategy is used for aggregating the old (lookup by its correlation id) and the new exchanges together into a single exchange. Possible implementations include performing some kind of combining or delta processing, such as adding line items together into an invoice or just using the newest exchange and removing old exchanges such as for state tracking or market data prices; where old values are of little use.

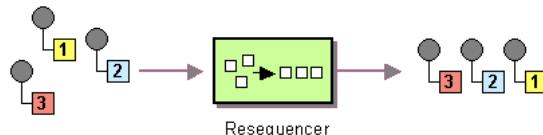
Notice the aggregation strategy is a mandatory option and must be provided to the aggregator.

Here are a few example AggregationStrategy implementations that should help you create your own custom strategy.

```
class ArrayListAggregationStrategy implements AggregationStrategy { public Exchange aggregate(Exchange oldExchange, Exchange newExchange) { Object newBody = newExchange.getIn().getBody(); ArrayList<Object> list = new ArrayList<Object>; list.add(newBody); newExchange.setBody(list); return newExchange; } }
```

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.



Change in Camel 2.7

The <batch-config> and <stream-config> tags in XML DSL in the Resequencer EIP must now be configured in the top, and not in the bottom. So if you use those, then move them up just below the <resequence> EIP starts in the XML. If you are using Camel older than 2.7, then those configs should be at the bottom.

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.

By default the Resequencer does not support duplicate messages and will only keep the last message, in case a message arrives with the same message expression. However in the batch mode you can enable it to allow duplicates.

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.

```
{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/ResequencerTest.java}This is equivalent to
from("direct:start").resequence(body()).batch().to("mock:result");
The batch-processing resequencer can be further configured via the size() and timeout() methods.
from("direct:start").resequence(body()).batch().size(300).timeout(4000L).to("mock:result")
This sets the batch size to 300 and the batch timeout to 4000 ms (by default, the batch size is 100 and the timeout is 1000 ms).
Alternatively, you can provide a configuration object.
from("direct:start").resequence(body()).batch(new BatchResequencerConfig(300, 4000L)).to("mock:result")
So the above example will reorder messages from endpoint direct:a in order of their bodies, to the endpoint mock:result.
Typically you'd use a header rather than the body to order things; or maybe a part of the body. So you could replace this
expression with
resequencer(header("mySeqNo"))
for example to reorder messages using a custom sequence number in the header mySeqNo.
```

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

Using the Spring XML Extensions

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

Allow Duplicates

Available as of Camel 2.4

In the batch mode, you can now allow duplicates. In Java DSL there is a `allowDuplicates()` method and in Spring XML there is an `allowDuplicates=true` attribute on the `<batch-config/>` you can use to enable it.

Reverse

Available as of Camel 2.4

In the batch mode, you can now reverse the expression ordering. By default the order is based on `0..9,A..Z`, which would let messages with low numbers be ordered first, and thus also outgoing first. In some cases you want to reverse order, which is now possible.

In Java DSL there is a `reverse()` method and in Spring XML there is an `reverse=true` attribute on the `<batch-config/>` you can use to enable it.

Resequence JMS messages based on JMSPriority

Available as of Camel 2.4

It's now much easier to use the Resequencer to resequence messages from JMS queues based on `JMSPriority`. For that to work you need to use the two new options `allowDuplicates` and `reverse`.

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-jms/src/test/java/org/apache/camel/component/jms/JmsBatchResequencerJMSPriorityTest.java}Notice this is only possible in the
batch mode of the Resequencer.
```

Ignore invalid exchanges

Available as of Camel 2.9

The Resequencer EIP will from Camel 2.9 onwards throw a `CamelExchangeException` if the incoming Exchange is not valid for the resequencer - ie. the expression cannot be evaluated, such as a missing header. You can use the option `ignoreInvalidExchanges` to ignore these exceptions which means the Resequencer will then skip the invalid Exchange.

```
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/ResequenceBatchIgnoreInvalidExchangesTest.java}This option is available for
both batch and stream resequencer.
```

Reject Old Exchanges

Available as of Camel 2.11

This option can be used to prevent out of order messages from being sent regardless of the event that delivered messages downstream (capacity, timeout, etc). If enabled using `rejectOld()`, the Resequencer will throw a `MessageRejectedException` when an incoming Exchange is "older" (based on the Comparator) than the last delivered message. This provides an extra level of control with regards to delayed message ordering.

```
from("direct:start").onException(MessageRejectedException.class).handled(true).to("mock:error").end()
.resequence(header("seqno")).stream().timeout(1000).rejectOld().to("mock:result");
```

This option is available for the stream resequencer only.

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 [snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/StreamResequencerTest.java] The stream-processing resequencer can be further configured via the capacity() and timeout() methods.
 from("direct:start").resequence(header("seqnum")).stream().capacity(5000).timeout(4000L).to("mock:result")
 This sets the resequencer's capacity to 5000 and the timeout to 4000 ms (by default, the capacity is 1000 and the timeout is 1000 ms). Alternatively, you can provide a configuration object.
 from("direct:start").resequence(header("seqnum")).stream(new StreamResequencerConfig(5000, 4000L)).to("mock:result")
 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 successor 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 a custom expression. [snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/resequencer/MyFileNameExpression.java]
 {snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/resequencer/ResequencerFileNameTest.java} or custom comparator via the comparator() method
 ExpressionResultComparator<Exchange> comparator = new MyComparator(); from("direct:start").resequence(header("seqnum")).stream().comparator(comparator).to("mock:result");
 or via a StreamResequencerConfig object.
 ExpressionResultComparator<Exchange> comparator = new MyComparator(); StreamResequencerConfig config = new StreamResequencerConfig(100, 1000L, comparator); from("direct:start").resequence(header("seqnum")).stream(config).to("mock:result");
Using the Spring XML Extensions
 xml<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring"> <route> <from uri="direct:start"/> <resequence> <simple>in.header.seqnum</simple> <to uri="mock:result" /> <stream-config capacity="5000" timeout="4000"/> </resequence> </route> </camelContext>

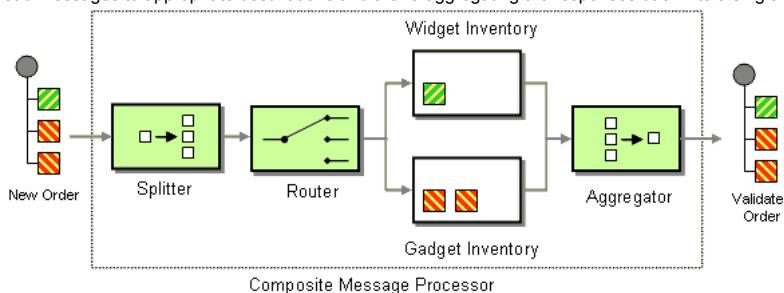
Further Examples

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

Composed Message Processor

The Composed Message Processor from the EIP patterns allows you to process a composite message by splitting it up, routing the sub-messages to appropriate destinations and the re-aggregating the responses back into a single message.



In Camel we provide two solutions

- using both a Splitter and Aggregator EIPs
- using only a Splitter

The difference is when using only a Splitter it aggregates back all the splitted messages into the same aggregation group, eg like a fork/join pattern.

Whereas using the Aggregator allows you group into multiple groups, a pattern which provides more options.

Using the splitter alone is often easier and possibly a better solution. So take a look at this first, before involving the aggregator.

Example using both Splitter and Aggregator

In this example we want to check that a multipart order can be filled. Each part of the order requires a check at a different inventory.

```
// split up the order so individual OrderItems can be validated by the appropriate bean
from("direct:start")
    .split().body()
    .choice()
        .when().method("orderItemHelper", "isWidget")
            .to("bean:widgetInventory")
        .otherwise()
            .to("bean:gadgetInventory")
    .end()
    .to("seda:aggregate");

// collect and re-assemble the validated OrderItems into an order again
from("seda:aggregate")
    .aggregate(new MyOrderAggregationStrategy()).header("orderId").completionTimeout(100).completionTimeoutCheckerInterval(10)
        .to("mock:result");
```

Using the Spring XML Extensions

```
<route>
    <from uri="direct:start"/>
    <split>
        <simple>body</simple>
        <choice>
            <when>
                <method bean="orderItemHelper" method="isWidget"/>
            <to uri="bean:widgetInventory"/>
            </when>
            <otherwise>
                <to uri="bean:gadgetInventory"/>
            </otherwise>
        </choice>
        <to uri="seda:aggregate"/>
    </split>
</route>

<route>
    <from uri="seda:aggregate"/>
    <aggregate strategyRef="myOrderAggregatorStrategy" completionTimeout="1000">
        <correlationExpression>
            <simple>header.orderId</simple>
        </correlationExpression>
        <to uri="mock:result"/>
    </aggregate>
</route>
```

To do this we split up the order using a Splitter. The Splitter then sends individual orderItems to a Content Based Router which checks the item type. Widget items get sent for checking in the widgetInventory bean and gadgets get sent to the gadgetInventory bean. Once these OrderItems have been validated by the appropriate bean, they are sent on to the Aggregator which collects and re-assembles the validated orderItems into an order again.

When an order is sent it contains a header with the order id. We use this fact when we aggregate, as we configure this .header("orderId") on the aggregate DSL to instruct Camel to use the header with the key orderId as correlation expression.

For full details, check the example source here:

[camel-core/src/test/java/org/apache/camel/processor/ComposedMessageProcessorTest.java](#)

Example using only Splitter

In this example we want to split an incoming order using the Splitter eip, transform each order line, and then combine the order lines into a new order message.

```
// this routes starts from the direct:start endpoint
// the body is then splitted based on @ separator
// the splitter in Camel supports InOut as well and for that we need
// to be able to aggregate what response we need to send back, so we provide our
// own strategy with the class MyOrderStrategy.
from("direct:start")
    .split(body().tokenize("@"), new MyOrderStrategy())
        // each splitted message is then send to this bean where we can process it
        .to("bean:MyOrderService?method=handleOrder")
        // this is important to end the splitted route as we do not want to do more routing
        // on each splitted message
    .end()
    // after we have splitted and handled each message we want to send a single combined
    // response back to the original caller, so we let this bean build it for us
    // this bean will receive the result of the aggregate strategy: MyOrderStrategy
    .to("bean:MyOrderService?method=buildCombinedResponse")
```

Using XML

If you use XML, then the <split> tag offers the strategyRef attribute to refer to your custom AggregationStrategy

The bean with the methods to transform the order line and process the order as well:

```
public static class MyOrderService {
    private static int counter;

    /**
     * We just handle the order by returning a id line for the order
     */
    public String handleOrder(String line) {
        LOG.debug("HandleOrder: " + line);
        return "(id=" + ++counter + ",item=" + line + ")";
    }

    /**
     * We use the same bean for building the combined response to send
     * back to the original caller
     */
    public String buildCombinedResponse(String line) {
        LOG.debug("BuildCombinedResponse: " + line);
        return "Response[" + line + "]";
    }
}
```

And the AggregationStrategy we use with the Splitter eip to combine the orders back again (eg fork/join):

```
/*
 * This is our own order aggregation strategy where we can control
 * how each splitted message should be combined. As we do not want to
 * loos any message we copy from the new to the old to preserve the
 * order lines as long we process them
 */
public static class MyOrderStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        // put order together in old exchange by adding the order from new exchange
        if (oldExchange == null) {
            // the first time we aggregate we only have the new exchange,
            // so we just return it
            return newExchange;
        }

        String orders = oldExchange.getIn().getBody(String.class);
        String newline = newExchange.getIn().getBody(String.class);

        LOG.debug("Aggregate old orders: " + orders);
        LOG.debug("Aggregate new order: " + newline);

        // put orders together separating by semi colon
        orders = orders + ":" + newline;
        // put combined order back on old to preserve it
        oldExchange.getIn().setBody(orders);

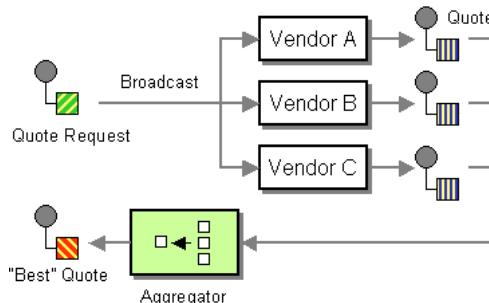
        // return old as this is the one that has all the orders gathered until now
        return oldExchange;
    }
}
```

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.

Scatter-Gather

The Scatter-Gather from the EIP patterns allows you to route messages to a number of dynamically specified recipients and re-aggregate the responses back into a single message.



Dynamic Scatter-Gather Example

In this example we want to get the best quote for beer from several different vendors. We use a dynamic Recipient List to get the request for a quote to all vendors and an Aggregator to pick the best quote out of all the responses. The routes for this are

defined as:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <recipientList>
      <header>listOfVendors</header>
    </recipientList>
  </route>
  <from uri="seda:quoteAggregator"/>
  <aggregate strategyRef="aggregatorStrategy" completionTimeout="1000">
    <correlationExpression>
      <header>quoteRequestId</header>
    </correlationExpression>
    <to uri="mock:result"/>
  </aggregate>
  </route>
</camelContext>
```

So in the first route you see that the Recipient List is looking at the `listOfVendors` header for the list of recipients. So, we need to send a message like

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("listOfVendors", "bean:vendor1, bean:vendor2, bean:vendor3");
headers.put("quoteRequestId", "quoteRequest-1");
template.sendBodyAndHeaders("direct:start", "<quote_request item=\"beer\"/>", headers);
```

This message will be distributed to the following Endpoints: `bean:vendor1`, `bean:vendor2`, and `bean:vendor3`. These are all beans which look like

```
public class MyVendor {
  private int beerPrice;

  @Produce(uri = "seda:quoteAggregator")
  private ProducerTemplate quoteAggregator;

  public MyVendor(int beerPrice) {
    this.beerPrice = beerPrice;
  }

  public void getQuote(@XPath("/quote_request/@item") String item, Exchange exchange) throws Exception {
    if ("beer".equals(item)) {
      exchange.getIn().setBody(beerPrice);
      quoteAggregator.send(exchange);
    } else {
      throw new Exception("No quote available for " + item);
    }
  }
}
```

and are loaded up in Spring like

```
<bean id="aggregatorStrategy" class="org.apache.camel.spring.processor.scattergather.LowestQuoteAggregationStrategy"/>
<bean id="vendor1" class="org.apache.camel.spring.processor.scattergather.MyVendor">
  <constructor-arg>
    <value>1</value>
  </constructor-arg>
</bean>

<bean id="vendor2" class="org.apache.camel.spring.processor.scattergather.MyVendor">
  <constructor-arg>
    <value>2</value>
  </constructor-arg>
</bean>

<bean id="vendor3" class="org.apache.camel.spring.processor.scattergather.MyVendor">
  <constructor-arg>
    <value>3</value>
  </constructor-arg>
</bean>
```

Each bean is loaded with a different price for beer. When the message is sent to each bean endpoint, it will arrive at the `MyVendor.getQuote` method. This method does a simple check whether this quote request is for beer and then sets the price of beer on the exchange for retrieval at a later step. The message is forwarded on to the next step using POJO Producing (see the `@Produce` annotation).

At the next step we want to take the beer quotes from all vendors and find out which one was the best (i.e. the lowest!). To do this we use an Aggregator with a custom aggregation strategy. The Aggregator needs to be able to compare only the messages from this particular quote; this is easily done by specifying a correlationExpression equal to the value of the `quoteRequestId` header. As shown above in the message sending snippet, we set this header to `quoteRequest-1`. This correlation value should be unique or you may include responses that are not part of this quote. To pick the lowest quote out of the set, we use a custom aggregation strategy like

```
public class LowestQuoteAggregationStrategy implements AggregationStrategy {
  public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
    // the first time we only have the new exchange
    if (oldExchange == null) {
      return newExchange;
    }

    if (oldExchange.getIn().getBody(int.class) < newExchange.getIn().getBody(int.class)) {
```

```

        return oldExchange;
    } else {
        return newExchange;
    }
}

```

Finally, we expect to get the lowest quote of \$1 out of \$1, \$2, and \$3.

```
result.expectedBodiesReceived(1); // expect the lowest quote
```

You can find the full example source here:

[camel-spring/src/test/java/org/apache/camel/spring/processor/scattergather/camel-spring/src/test/resources/org/apache/camel/spring/processor/scattergather/scatter-gather.xml](#)

Static Scatter-Gather Example

You can lock down which recipients are used in the Scatter-Gather by using a static Recipient List. It looks something like this

```

from("direct:start").multicast().to("seda:vendor1", "seda:vendor2", "seda:vendor3");

from("seda:vendor1").to("bean:vendor1").to("seda:quoteAggregator");
from("seda:vendor2").to("bean:vendor2").to("seda:quoteAggregator");
from("seda:vendor3").to("bean:vendor3").to("seda:quoteAggregator");

from("seda:quoteAggregator")
    .aggregate(header("quoteRequestId"), new LowestQuoteAggregationStrategy()).to("mock:result")

```

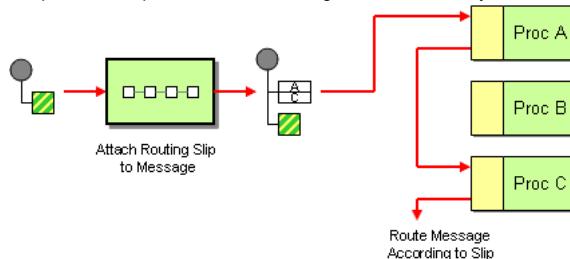
A full example of the static Scatter-Gather configuration can be found in the [Loan Broker Example](#).

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.

Routing Slip

The Routing Slip from the EIP patterns allows you to route a message consecutively through a series of processing steps where the sequence of steps is not known at design time and can vary for each message.



Options

confluenceTableSmall

Name	Default Value	Description
uriDelimiter	,	Delimiter used if the Expression returned multiple endpoints.
ignoreInvalidEndpoints	false	If an endpoint URI could not be resolved, should it be ignored. Otherwise, Camel will throw an exception stating the endpoint URI is not valid.
cacheSize	1000	Camel 2.13.1/2.12.4: Allows to configure the cache size for the ProducerCache which caches producers for reuse in the routing slip. The default cache size is 1000. A value of -1 disables the use of the cache.

Example

The following route will take any messages sent to the Apache ActiveMQ queue `Somequeue` and pass them into the Routing Slip pattern.

```
javafrom("activemq:SomeQueue") .routingSlip("aRoutingSlipHeader");
```

Messages will be checked for the existence of the `a:routingslipHeader` header. The value of this header should be a comma-delimited list of endpoint URLs you wish the message to be routed to. The Message will be routed in a pipeline fashion, i.e., one after the other. From **Camel 2.5** the Routing Slip will set a property, `Exchange.SLIP_ENDPOINT`, on the Exchange which contains the current endpoint as it advanced through the slip. This allows you to know how far we have processed in the slip.

The Routing Slip will compute the slip **beforehand** which means, the slip is only computed once. If you need to compute the slip *on-the-fly* then use the Dynamic Router pattern instead.

Configuration Options

Here we set the header name and the URI delimiter to something different.

```
Using the Fluent Builders{snippet:id=e3|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/routingslip/RoutingSlipTest.java}Using the Spring XML Extensions
<camelContext id="buildRoutingSlip" xmlns="http://activemq.apache.org/camel/schema/spring"> <route> <from uri="direct:c"/>
<routingSlip uriDelimiter="#"> <header>aRoutingSlipHeader</header> </routingSlip> </route> </camelContext>
```

Ignore Invalid Endpoints

Available as of Camel 2.3

The Routing Slip now supports `ignoreInvalidEndpoints` which the Recipient List also supports. You can use it to skip endpoints which are invalid.

```
javafrom("direct:a") .routingSlip("myHeader") .ignoreInvalidEndpoints();
```

And in Spring XML its an attribute on the recipient list tag:

```
xml<route> <from uri="direct:a"/> <routingSlip ignoreInvalidEndpoints="true"> <header>myHeader</header> </routingSlip>
</route>
```

Then let's say the `myHeader` contains the following two endpoints `direct:foo,xxx:bar`. The first endpoint is valid and works. However the second endpoint is invalid and will just be ignored. Camel logs at `INFO` level, so you can see why the endpoint was invalid.

Expression Support

Available as of Camel 2.4

The Routing Slip now supports to take the expression parameter as the Recipient List does. You can tell Camel the expression that you want to use to get the routing slip.

```
javafrom("direct:a") .routingSlip(header("myHeader")) .ignoreInvalidEndpoints();
```

And in Spring XML its an attribute on the recipient list tag.

```
xml<route> <from uri="direct:a"/> <!--NOTE from Camel 2.4.0, you need to specify the expression element inside of the
routingSlip element --> <routingSlip ignoreInvalidEndpoints="true"> <header>myHeader</header> </routingSlip> </route>
```

Further Examples

For further examples of this pattern in use you could look at the routing slip test cases.

Using This Pattern

Throttler

The Throttler Pattern allows you to ensure that a specific endpoint does not get overloaded, or that we don't exceed an agreed SLA with some external service.

Options

confluenceTableSmall

Name	Default Value	Description
<code>maximumRequestsPerPeriod</code>		Maximum number of requests per period to throttle. This option must be provided as a positive number. Notice, in the XML DSL, from Camel 2.8 onwards this option is configured using an Expression instead of an attribute.
<code>timePeriodMillis</code>	1000	The time period in milliseconds, in which the throttler will allow at most <code>maximumRequestsPerPeriod</code> number of messages.
<code>asyncDelayed</code>	false	Camel 2.4: If enabled then any messages which is delayed happens asynchronously using a scheduled thread pool.
<code>executorServiceRef</code>		Camel 2.4: Refers to a custom Thread Pool to be used if <code>asyncDelayed</code> has been enabled.
<code>callerRunswhenRejected</code>	true	Camel 2.4: Is used if <code>asyncDelayed</code> was enabled. This controls if the caller thread should execute the task if the thread pool rejected the task.
<code>rejectExecution</code>	false	Camel 2.14: If this option is true, throttler throws a

ThrottlerRejectExecutionException when the request rate exceeds the limit.

Examples

Using the Fluent Builders

```
{snippet:id=ex|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/ThrottlerTest.java}
So the above example will throttle messages all messages received on seda:a before being sent to mock:result ensuring that a maximum of 3 messages are sent in any 10 second window.
```

Note that since `timePeriodWith111's` defaults to 1000 milliseconds, just setting the `maximumRequestsPerPeriod` has the effect of setting the maximum number of requests per second. So to throttle requests at 100 requests per second between two endpoints, it would look more like this...

```
from("seda:a").throttle(100).to("seda:b");
```

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

Using the Spring XML Extensions

Camel 2.7.x or older

```
{snippet:id=example|lang=xml|url=camel/tags/camel-2.7.0/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/throttler.xml}
```

Camel 2.8 onwards

In Camel 2.8 onwards you must set the maximum period as an Expression as shown below where we use a Constant expression:

```
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/throttler.xml}
```

Dynamically changing maximum requests per period

Available as of Camel 2.8

Since we use an Expression you can adjust this value at runtime, for example you can provide a header with the value. At runtime Camel evaluates the expression and converts the result to a `java.lang.Long` type. In the example below we use a header from the message to determine the maximum requests per period. If the header is absent, then the Throttler uses the old value. So that allows you to only provide a header if the value is to be changed:

```
{snippet:id=e2|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/throttler.xml}
```

Asynchronous delaying

Available as of Camel 2.4

You can let the Throttler use non blocking asynchronous delaying, which means Camel will use a scheduler to schedule a task to be executed in the future. The task will then continue routing. This allows the caller thread to not block and be able to service other messages, etc.

```
from("seda:a").throttle(100).asyncDelayed().to("seda:b");
```

Using This Pattern

SAMPLING THROTTLER

Available as of Camel 2.1

A sampling throttler allows you to extract a sample of the exchanges from the traffic through a route.

It is configured with a sampling period during which only a single exchange is allowed to pass through. All other exchanges will be stopped.

Will by default use a sample period of 1 seconds.

Options

confluenceTableSmall

Name	Default Value	Description
messageFrequency		Samples the message every N'th message. You can only use either frequency or period.
samplePeriod	1	Samples the message every N'th period. You can only use either frequency or period.
units	SECOND	Time unit as an enum of <code>java.util.concurrent.TimeUnit</code> from the JDK.

Samples

You use this EIP with the sample DSL as show in these samples.

Using the Fluent Builders

These samples also show how you can use the different syntax to configure the sampling period:

```
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/SamplingThrottlerTest.java}
```

Using the Spring XML Extensions

And the same example in Spring XML is:

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/samplingThrottler.xml}
```

And since it uses a default of 1 second you can omit this configuration in case you also want to use 1 second

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/samplingThrottlerWithDefault.xml} Using This Pattern
```

See Also

- Throttler
- Aggregator

Delayer

The Delayer Pattern allows you to delay the delivery of messages to some destination.

Delayer

The expression is a value in millis to wait from the current time, so the expression should just be 3000.

However you can use a long value for a fixed value to indicate the delay in millis.

See the Spring DSL samples for Delayer.

Using Delayer in Java DSL

See this ticket: <https://issues.apache.org/jira/browse/CAMEL-2654>

Options

confluenceTableSmall

Name	Default Value	Description
asyncDelayed	false	Camel 2.4: If enabled then delayed messages happens asynchronously using a scheduled thread pool.
executorServiceRef		Camel 2.4: Refers to a custom Thread Pool to be used if <code>asyncDelayed</code> has been enabled.
callerRunsWhenRejected	true	Camel 2.4: Is used if <code>asyncDelayed</code> was enabled. This controls if the caller thread should execute the task if the thread pool rejected the task.

Using the Fluent Builders

The example below will delay all messages received on `seda:b` 1 second before sending them to `mock:result`.

```
{snippet:id=ex2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/DelayerTest.java}
```

You can just delay things a fixed amount of time from the point at which the delayer receives the message. For example to delay things 2 seconds

```
delayer(2000)
```

The above assume that the delivery order is maintained and that the messages are delivered in delay order. If you want to reorder the messages based on delivery time, you can use the Resequencer with this pattern. For example

```
from("activemq:someQueue").resequencer(header("MyDeliveryTime")).delay("MyRedeliveryTime").to("activemq:aDelayedQueue");
```

You can of course use many different Expression languages such as XPath, XQuery, SQL or various Scripting Languages. For example to delay the message for the time period specified in the header, use the following syntax:

```
from("activemq:someQueue").delay(header("delayValue")).to("activemq:aDelayedQueue");
```

And to delay processing using the Simple language you can use the following DSL:

```
from("activemq:someQueue").delay(simple("${body.delayProperty}")).to("activemq:aDelayedQueue");
```

Spring DSL

The sample below demonstrates the delay in Spring DSL:

```
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/delayer.xml}
```

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

Asynchronous delaying

Available as of Camel 2.4

You can let the Delayer use non blocking asynchronous delaying, which means Camel will use a scheduler to schedule a task to be executed in the future. The task will then continue routing. This allows the caller thread to not block and be able to service other messages etc.

From Java DSL

You use the `asyncDelayed()` to enable the async behavior.
`from("activemq:queue:foo").delay(1000).asyncDelayed().to("activemq:aDelayedQueue");`

From Spring XML

You use the `asyncDelayed="true"` attribute to enable the async behavior.
`<xml-route> <from uri="activemq:queue:foo"/> <delay asyncDelayed="true"> <constant>1000</constant> </delay> <to uri="activemq:aDelayedQueue"/> </route>`

Creating a custom delay

You can use an expression to determine when to send a message using something like this
`from("activemq:foo").delay().method("someBean", "computeDelay").to("activemq:bar");`
then the bean would look like this...
`public class SomeBean { public long computeDelay() { long delay = 0; // use java code to compute a delay value in millis return delay; } }`
Using This Pattern

See Also

- Delay Interceptor

Load Balancer

The Load Balancer Pattern allows you to delegate to one of a number of endpoints using a variety of different load balancing policies.

Built-in load balancing policies

Camel provides the following policies out-of-the-box:

Policy	Description
Round Robin	The exchanges are selected from in a round robin fashion. This is a well known and classic policy, which spreads the load evenly.
Random	A random endpoint is selected for each exchange.
Sticky	Sticky load balancing using an Expression to calculate a correlation key to perform the sticky load balancing; rather like <code>jsessionid</code> in the web or <code>JMSXGroupID</code> in JMS.
Topic	Topic which sends to all destinations (rather like JMS Topics)
Failover	In case of failures the exchange will be tried on the next endpoint.
Weighted Round-Robin	Camel 2.5: The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to the others. In addition to the weight, endpoint selection is then further refined using round-robin distribution based on weight.
Weighted Random	Camel 2.5: The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to others. In addition to the weight, endpoint selection is then further refined using random distribution based on weight.
Custom	Camel 2.8: From Camel 2.8 onwards the preferred way of using a custom Load Balancer is to use this policy, instead of using the <code>@deprecated ref</code> attribute.
Circuit Breaker	Camel 2.14: Implements the Circuit Breaker pattern as described in "Release it!" book.

Load balancing HTTP endpoints

If you are proxying and load balancing HTTP, then see this page for more details.

Round Robin

The round robin load balancer is not meant to work with failover, for that you should use the dedicated **failover** load balancer. The round robin load balancer will only change to next endpoint per message.

The round robin load balancer is stateful as it keeps state of which endpoint to use next time.

Using the Fluent Builders{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/RoundRobinLoadBalanceTest.java}Using the Spring configuration
 xml<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring"> <route> <from uri="direct:start"/> <loadBalance> <roundRobin/> <to uri="mock:x"/> <to uri="mock:y"/> <to uri="mock:z"/> </loadBalance> </route> </camelContext>
 The above example loads balance requests from **direct:start** to one of the available **mock endpoint** instances, in this case using a round robin policy.

For further examples of this pattern look at this junit test case

Failover

The failover load balancer is capable of trying the next processor in case an Exchange failed with an exception during processing.

You can constrain the failover to activate only when one exception of a list you specify occurs. If you do not specify a list any exception will cause fail over to occur. This balancer uses the same strategy for matching exceptions as the Exception Clause does for the **onException**.

Enable stream caching if using streams

If you use streaming then you should enable Stream caching when using the failover load balancer. This is needed so the stream can be re-read after failing over to the next processor.

Failover offers the following options:

Option	Type	Default	Description
inheritErrorHandler	boolean	true	Camel 2.3: Whether or not the Error Handler configured on the route should be used. Disable this if you want failover to transfer immediately to the next endpoint. On the other hand, if you have this option enabled, then Camel will first let the Error Handler try to process the message. The Error Handler may have been configured to redeliver and use delays between attempts. If you have enabled a number of redeliveries then Camel will try to redeliver to the same endpoint , and only fail over to the next endpoint, when the Error Handler is exhausted.
maximumFailoverAttempts	int	-1	Camel 2.3: A value to indicate after X failover attempts we should exhaust (give up). Use -1 to indicate never give up and continuously try to failover. Use 0 to never failover. And use e.g. 3 to failover at most 3 times before giving up. This option can be used whether or not roundRobin is enabled or not.
roundRobin	boolean	false	Camel 2.3: Whether or not the failover load balancer should operate in round robin mode or not. If not, then it will always start from the first endpoint when a new message is to be processed. In other words it restart from the top for every message. If round robin is enabled, then it keeps state and will continue with the next endpoint in a round robin fashion. When using round robin it will not stick to last known good endpoint, it will always pick the next endpoint to use. You can also enable sticky mode together with round robin, if so then it will pick the last known good endpoint to use when starting the load balancing (instead of using the next when starting).
sticky	boolean	false	Camel 2.16: Whether or not the failover load balancer should operate in sticky mode or not. If not, then it will always start from the first endpoint when a new message is to be processed. In other words it restart from the top for every message. If sticky is enabled, then it keeps state and will continue with the last known good endpoint. You can also enable sticky mode together with round robin, if so then it will pick the last known good endpoint to use when starting the load balancing (instead of using the next when starting).

Camel 2.2 or older behavior

The current implementation of failover load balancer uses simple logic which **always** tries the first endpoint, and in case of an exception being thrown it tries the next in the list, and so forth. It has no state, and the next message will thus **always** start with the first endpoint.

Camel 2.3 onwards behavior

The failover load balancer now supports round robin mode, which allows you to failover in a round robin fashion. See the **roundRobin** option.

Redelivery must be enabled

In Camel 2.2 or older the failover load balancer requires you have enabled Camel Error Handler to use redelivery. In Camel 2.3 onwards this is not required as such, as you can mix and match. See the **inheritErrorHandler** option.

Here is a sample to failover only if a IOException related exception was thrown:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/FailOverNotCatchedExceptionTest.java}You can specify multiple exceptions to failover as the option is varargs, for instance:

```
java// enable redelivery so failover can react
errorHandler(defaultErrorHandler().maximumRedeliveries(5));
from("direct:foo").loadBalance().failover(IOException.class,
MyOtherException.class).to("direct:a", "direct:b");
```

Using failover in Spring DSL

Failover can also be used from Spring DSL and you configure it as:

```
xml <route errorHandlerRef="myErrorHandler"> <from uri="direct:foo"/> <loadBalance> <failover>
<exception>java.io.IOException</exception> <exception>com.mycompany.MyOtherException</exception> </failover> <to
uri="direct:a"/> <to uri="direct:b"/> </loadBalance> </route>
```

Using failover in round robin mode

An example using Java DSL:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/FailoverRoundRobinTest.java}And the same example using Spring XML:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/FailoverRoundRobinTest.xml}

Disabled inheritErrorHandler

You can configure `inheritErrorHandler=false` if you want to failover to the next endpoint as fast as possible. By disabling the Error Handler you ensure it does not *intervene* which allows the failover load balancer to handle failover asap. By also enabling `roundRobin` mode, then it will keep retrying until it success. You can then configure the `maximumFailoverAttempts` option to a high value to let it eventually exhaust (give up) and fail.

Weighted Round-Robin and Random Load Balancing

Available as of Camel 2.5

In many enterprise environments where server nodes of unequal processing power & performance characteristics are utilized to host services and processing endpoints, it is frequently necessary to distribute processing load based on their individual server capabilities so that some endpoints are not unfairly burdened with requests. Obviously simple round-robin or random load balancing do not alleviate problems of this nature. A Weighted Round-Robin and/or Weighted Random load balancer can be used to address this problem.

The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to others. You can specify this as a positive processing weight for each server. A larger number indicates that the server can handle a larger load. The weight is utilized to determine the payload distribution ratio to different processing endpoints with respect to others.

Disabled inheritErrorHandler

As of Camel 2.6, the Weighted Load balancer usage has been further simplified, there is no need to send in `distributionRatio` as a `List<Integer>`. It can be simply sent as a delimited String of integer weights separated by a delimiter of choice.

The parameters that can be used are

In Camel 2.5

Option	Type	Default	Description
roundRobin	boolean	false	The default value for round-robin is false. In the absence of this setting or parameter the load balancing algorithm used is random.
distributionRatio	List<Integer>	none	The distributionRatio is a list consisting on integer weights passed in as a parameter. The distributionRatio must match the number of endpoints and/or processors specified in the load balancer list. In Camel 2.5 if endpoints do not match ratios, then a best effort distribution is attempted.

Available In Camel 2.6

Option	Type	Default	Description
roundRobin	boolean	false	The default value for round-robin is false. In the absence of this setting or parameter the load balancing algorithm used is random.
distributionRatio	String	none	The distributionRatio is a delimited String consisting on integer weights separated by delimiters for example "2,3,5". The distributionRatio must match the number of endpoints and/or processors specified in the load balancer list.
distributionRatioDelimiter	String	,	The distributionRatioDelimiter is the delimiter used to specify the distributionRatio. If this attribute is not specified a default delimiter "," is expected as the delimiter used for specifying the distributionRatio.

Using Weighted round-robin & random load balancing

In Camel 2.5

An example using Java DSL:

```
javaArrayList<integer> distributionRatio = new ArrayList<integer>(); distributionRatio.add(4); distributionRatio.add(2);
distributionRatio.add(1); // round-robin from("direct:start") .loadBalance().weighted(true, distributionRatio) .to("mock:x", "mock:y",
"mock:z"); //random from("direct:start") .loadBalance().weighted(false, distributionRatio) .to("mock:x", "mock:y", "mock:z");
And the same example using Spring XML:
xml <route> <from uri="direct:start"/> <loadBalance> <weighted roundRobin="false" distributionRatio="4 2 1"/> <to uri="mock:x"/>
<to uri="mock:y"/> <to uri="mock:z"/> </loadBalance> </route>
```

Available In Camel 2.6

An example using Java DSL:

```
java:// round-robin from("direct:start") .loadBalance().weighted(true, "4:2:1" distributionRatioDelimiter=".") .to("mock:x", "mock:y", "mock:z"); //random from("direct:start") .loadBalance().weighted(false, "4,2,1") .to("mock:x", "mock:y", "mock:z");
And the same example using Spring XML:
xml <route> <from uri="direct:start"/> <loadBalance> <weighted roundRobin="false" distributionRatio="4-2-1"
distributionRatioDelimiter="-" /> <to uri="mock:x"/> <to uri="mock:y"/> <to uri="mock:z"/> </loadBalance> </route>
```

Custom Load Balancer

You can use a custom load balancer (eg your own implementation) also.

An example using Java DSL:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/CustomLoadBalanceTest.java}And the same example using XML DSL:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringCustomRefLoadBalanceTest.xml}Notice in the XML DSL above we use <custom> which is only available in **Camel 2.8** onwards. In older releases you would have to do as follows instead:
xml <loadBalance ref="myBalancer"> <!-- these are the endpoints to balancer --> <to uri="mock:x"/> <to uri="mock:y"/> <to uri="mock:z"/> </loadBalance>

To implement a custom load balancer you can extend some support classes such as `LoadBalancerSupport` and `SimpleLoadbalancerSupport`. The former supports the asynchronous routing engine, and the latter does not. Here is an example:{snippet:id=e2|title=Custom load balancer implementation|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/CustomLoadBalanceTest.java}

Circuit Breaker

The Circuit Breaker load balancer is a stateful pattern that monitors all calls for certain exceptions. Initially the Circuit Breaker is in closed state and passes all messages. If there are failures and the threshold is reached, it moves to open state and rejects all calls until halfOpenAfter timeout is reached. After this timeout is reached, if there is a new call, it will pass and if the result is success the Circuit Breaker will move to closed state, or to open state if there was an error.

When the circuit breaker is closed, it will throw a `java.util.concurrent.RejectedExecutionException`. This can then be caught to provide an alternate path for processing exchanges.

An example using Java DSL:

```
javafrom("direct:start") .onException(RejectedExecutionException.class) .handled(true) .to("mock:serviceUnavailable") .end()
.loadBalance() .circuitBreaker(2, 1000L, MyCustomException.class) .to("mock:service") .end();
```

And the same example using Spring XML:

```
<camelize id="camel" xmlns="http://camel.apache.org/schema/spring"> <route> <from uri="direct:start"/> <onException>
<exception>java.util.concurrent.RejectedExecutionException</exception> <handled><constant>true</constant></handled> <to
uri="mock:serviceUnavailable"/> </onException> <loadBalance> <circuitBreaker threshold="2" halfOpenAfter="1000">
<exception>MyCustomException</exception> </circuitBreaker> <to uri="mock:service"/> </loadBalance> </route>
</camelize>
```

Using This Pattern

Multicast

The Multicast allows to route the same message to a number of endpoints and process them in a different way. The main difference between the Multicast and Splitter is that Splitter will split the message into several pieces but the Multicast will not modify the request message.

Options

confluenceTableSmall

Name	Default Value	Description
strategyRef		Refers to an AggregationStrategy to be used to assemble the replies from the multicasts, into a single outgoing message from the Multicast. By default Camel will use the last reply as the outgoing message. From Camel 2.12 onwards you can also use a POJO as the AggregationStrategy, see the Aggregate page for more details. If an exception is thrown from the aggregate method in the AggregationStrategy, then by default, that exception is not handled by the error handler. The error handler can be enabled to react if enabling the shareUnitOfWork option.
strategyMethodName		Camel 2.12: This option can be used to explicit declare the method name to use, when using POJOs as the AggregationStrategy. See the Aggregate page for more details.
strategyMethodAllowNull	false	Camel 2.12: If this option is false then the aggregate method is not used if there was no data to enrich. If this option is true then null values is used as

		the oldExchange (when no data to enrich), when using POJOs as the AggregationStrategy. See the Aggregate page for more details.
parallelProcessing	false	If enabled then sending messages to the multicasts occurs concurrently. Note the caller thread will still wait until all messages has been fully processed, before it continues. Its only the sending and processing the replies from the multicasts which happens concurrently.
parallelAggregate	false	Camel 2.14: If enabled then the aggregate method on AggregationStrategy can be called concurrently. Notice that this would require the implementation of AggregationStrategy to be implemented as thread-safe. By default this is false meaning that Camel synchronizes the call to the aggregate method. Though in some use-cases this can be used to archive higher performance when the AggregationStrategy is implemented as thread-safe.
executorServiceRef		Refers to a custom Thread Pool to be used for parallel processing. Notice if you set this option, then parallel processing is automatic implied, and you do not have to enable that option as well.
stopOnException	false	Camel 2.2: Whether or not to stop continue processing immediately when an exception occurred. If disable, then Camel will send the message to all multicasts regardless if one of them failed. You can deal with exceptions in the AggregationStrategy class where you have full control how to handle that.
streaming	false	If enabled then Camel will process replies out-of-order, eg in the order they come back. If disabled, Camel will process replies in the same order as multicasted.
timeout		Camel 2.5: Sets a total timeout specified in millis. If the Multicast hasn't been able to send and process all replies within the given timeframe, then the timeout triggers and the Multicast breaks out and continues. Notice if you provide a TimeoutAwareAggregationStrategy then the timeout method is invoked before breaking out. If the timeout is reached with running tasks still remaining, certain tasks for which it is difficult for Camel to shut down in a graceful manner may continue to run. So use this option with a bit of care. We may be able to improve this functionality in future Camel releases.
onPrepareRef		Camel 2.8: Refers to a custom Processor to prepare the copy of the Exchange each multicast will receive. This allows you to do any custom logic, such as deep-cloning the message payload if that's needed etc.
shareUnitOfWork	false	Camel 2.8: Whether the unit of work should be shared. See the same option on Splitter for more details.

Example

The following example shows how to take a request from the `direct:a` endpoint , then multicast these request to `direct:x`, `direct:y`, `direct:z`.

```
Using the Fluent Builders{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/MulticastTest.java}By default Multicast invokes each endpoint sequentially. If parallel processing is desired, simply use
from("direct:a").multicast().parallelProcessing().to("direct:x", "direct:y", "direct:z");
In case of using InOut MEP, an AggregationStrategy is used for aggregating all reply messages. The default is to only use the latest reply message and discard any earlier replies. The aggregation strategy is configurable:
from("direct:start") .multicast(new MyAggregationStrategy()) .parallelProcessing().timeout(500).to("direct:a", "direct:b", "direct:c")
.end() .to("mock:result");
```

Stop processing in case of exception

Available as of Camel 2.1

The Multicast will by default continue to process the entire Exchange even in case one of the multicasted messages will thrown an exception during routing.

For example if you want to multicast to 3 destinations and the 2nd destination fails by an exception. What Camel does by default is to process the remainder destinations. You have the chance to remedy or handle this in the AggregationStrategy.

But sometimes you just want Camel to stop and let the exception be propagated back, and let the Camel error handler handle it. You can do this in Camel 2.1 by specifying that it should stop in case of an exception occurred. This is done by the stopOnException option as shown below:

```
from("direct:start") .multicast().stopOnException().to("direct:foo", "direct:bar", "direct:baz") .end() .to("mock:result");
from("direct:foo").to("mock:foo"); from("direct:bar").process(new MyProcessor()).to("mock:bar"); from("direct:baz").to("mock:baz");
And using XML DSL you specify it as follows:
xml <route> <from uri="direct:start"/> <multicast stopOnException="true"> <to uri="direct:foo"/> <to uri="direct:bar"/> <to uri="direct:baz"/> </multicast> <to uri="mock:result"/> </route> <route> <from uri="direct:foo"/> <to uri="mock:foo"/> </route>
<route> <from uri="direct:bar"/> <process ref="myProcessor"/> <to uri="mock:bar"/> </route> <route> <from uri="direct:baz"/> <to uri="mock:baz"/> </route>
```

Using onPrepare to execute custom logic when preparing messages

Available as of Camel 2.8

The Multicast will copy the source Exchange and multicast each copy. However the copy is a shallow copy, so in case you have mutable message bodies, then any changes will be visible by the other copied messages. If you want to use a deep clone copy then you need to use a custom onPrepare which allows you to do this using the Processor interface.

Notice the onPrepare can be used for any kind of custom logic which you would like to execute before the Exchange is being multicasted.

Design for immutable

Its best practice to design for immutable objects.

For example if you have a mutable message body as this Animal class:

```
{snippet:id=e1|lang=java|title=Animal|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/Animal.java}Then we can create a deep clone processor which clones the message body:
```

```
{snippet:id=e1|lang=java|title=AnimalDeepClonePrepare|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/AnimalDeepClonePrepare.java}Then we can use the AnimalDeepClonePrepare class in the Multicast route using the onPrepare option as shown:{snippet:id=e1|lang=java|title=Multicast using onPrepare|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/MulticastOnPrepareTest.java}And the same example in XML DSL.{snippet:id=e1|lang=xml|title=Multicast using onPrepare|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/MulticastOnPrepareTest.xml}Notice the onPrepare option is also available on other EIPs such as Splitter, Recipient List, and Wire Tap.
```

Using This Pattern

LOOP

The Loop allows for processing a message a number of times, possibly in a different way for each iteration. Useful mostly during testing.

Default mode

Notice by default the loop uses the same exchange throughout the looping. So the result from the previous iteration will be used for the next (eg Pipes and Filters). From **Camel 2.8** onwards you can enable copy mode instead. See the options table for more details.

Options

confluenceTableSmall

Name	Default Value	Description
copy	false	Camel 2.8: Whether or not copy mode is used. If false then the same Exchange will be used for each iteration. So the result from the previous iteration will be visible for the next iteration. Instead you can enable copy mode, and then each iteration restarts with a fresh copy of the input Exchange.
doWhile		Camel 2.17: Enables the while loop that loops until the predicate evaluates to false or null.

Exchange properties

For each iteration two properties are set on the Exchange. Processors can rely on these properties to process the Message in different ways.

Property	Description
CamelLoopSize	Total number of loops. This is not available if running the loop in while loop mode.
CamelLoopIndex	Index of the current iteration (0 based)

Examples

The following example shows how to take a request from the **direct:x** endpoint, then send the message repetitively to **mock:result**. The number of times the message is sent is either passed as an argument to `loop()`, or determined at runtime by evaluating an expression. The expression **must** evaluate to an `int`, otherwise a `RuntimeCamelException` is thrown.

Using the Fluent Builders

Pass loop count as an argument{snippet:id=ex1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/LoopTest.java}Use expression to determine loop count{snippet:id=ex2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/LoopTest.java}Use expression to determine loop count{snippet:id=ex3|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/LoopTest.java}Using the Spring XML Extensions

Pass loop count as an argument{snippet:id=ex1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/loop.xml}Use expression to determine loop count{snippet:id=ex2|lang=xml|url=camel/trunk/components/camel-

spring/src/test/resources/org/apache/camel/spring/processor/loop.xml}For further examples of this pattern in use you could look at one of the junit test case

Using copy mode

Available as of Camel 2.8

Now suppose we send a message to "direct:start" endpoint containing the letter A. The output of processing this route will be that, each "mock:loop" endpoint will receive "AB" as message.
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/LoopCopyTest.java}However if we do **not** enable copy mode then "mock:loop" will receive "AB", "ABB", "ABBB", etc. messages.
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/LoopNoCopyTest.java}The equivalent example in XML DSL in copy mode is as follows:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringLoopCopyTest.xml}

Using while mode

Available as of Camel 2.17

The loop can act like a while loop that loops until the expression evaluates to false or null. For example the route below loops while the length of the message body is 5 or less characters. Notice that the DSL uses **loopDoWhile**.

```
from("direct:start") .loopDoWhile(simple("${body.length} <= 5")) .to("mock:loop") .transform(body().append("A")) .end() .to("mock:result");
```

And the same example in XML:

```
<route> <from uri="direct:start"/> <loop doWhile="true"> <simple>${body.length} &lt;= 5</simple> <to uri="mock:loop"/> <transform> <simple>A${body}</simple> </transform> </loop> <to uri="mock:result"/> </route>
```

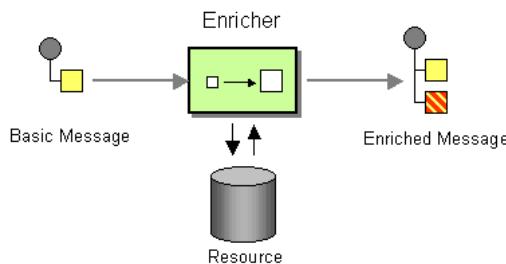
Notice in XML that the while loop is turned on using the **doWhile** attribute.

Using This Pattern

MESSAGE TRANSFORMATION

Content Enricher

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



Content enrichment using a Message Translator or a Processor

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{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/TransformViaDSLTest.java}In this example we add our own Processor using explicit Java code{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/TransformTest.java}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>
enrich-dsl
```

Content enrichment using the `enrich` DSL element

Camel comes with two flavors of content enricher in the DSL

- `enrich`
- `pollEnrich`

`enrich` uses a Producer to obtain the additional data. It is usually used for Request Reply messaging, for instance to invoke an external web service.

`pollEnrich` on the other hand uses a Polling Consumer to obtain the additional data. It is usually used for Event Message messaging, for instance to read a file or download a FTP file.

Camel 2.15 or older - Data from current Exchange not used

`pollEnrich` or `enrich` does **not** access any data from the current Exchange which means when polling it cannot use any of the existing headers you may have set on the Exchange. For example you cannot set a filename in the `Exchange.FILE_NAME` header and use `pollEnrich` to consume only that file. For that you **must** set the filename in the endpoint URI.

Instead of using `enrich` you can use Recipient List and have dynamic endpoints and define an `AggregationStrategy` on the Recipient List which then would work as a `enrich` would do.

`pollEnrich` only accept one message as response. That means that if you target to enrich your original message with the enricher collecting messages from a seda, ... components using an aggregation strategy. Only one response message will be aggregated with the original message.

From **Camel 2.16** onwards both `enrich` and `pollEnrich` supports dynamic endpoints that uses an Expression to compute the uri, which allows to use data from the current Exchange. In other words all what is told above no longer apply and it just works.

Enrich Options

confluenceTableSmall

Name	Default Value	Description
<code>uri</code>		The endpoint uri for the external service to enrich from. You must use either <code>uri</code> or <code>ref</code> . Important: From Camel 2.16 onwards, this option is removed, and you use an Expression to configure the uri, such as Simple or Constant or any other dynamic language that can compute the uri dynamically using values from the current Exchange.
<code>ref</code>		Refers to the endpoint for the external service to enrich from. You must use either <code>uri</code> or <code>ref</code> . Important: From Camel 2.16 onwards, this option is removed, and you use an Expression to configure the uri, such as Simple or Constant or any other dynamic language that can compute the uri dynamically using values from the current Exchange.
<code>expression</code>		Camel 2.16: Mandatory. The Expression to configure the uri, such as Simple or Constant or any other dynamic language that can compute the uri dynamically using values from the current Exchange.
<code>strategyRef</code>		Refers to an <code>AggregationStrategy</code> to be used to merge the reply from the external service, into a single outgoing message. By default Camel will use the reply from the external service as outgoing message. From Camel 2.12 onwards you can also use a POJO as the <code>AggregationStrategy</code> , see the Aggregate page for more details.
<code>strategyMethodName</code>		Camel 2.12: This option can be used to explicit declare the method name to use, when using POJOs as the <code>AggregationStrategy</code> . See the Aggregate page for more details.
<code>strategyMethodAllowNull</code>	<code>false</code>	Camel 2.12: If this option is <code>false</code> then the aggregate method is not used if there was no data to enrich. If this option is <code>true</code> then <code>null</code> values is used as the <code>oldexchange</code> (when no data to enrich), when using POJOs as the <code>AggregationStrategy</code> . See the Aggregate page for more details.
<code>aggregateOnException</code>	<code>false</code>	Camel 2.14: If this option is <code>false</code> then the aggregate method is not used if there was an exception thrown while trying to retrieve the data to enrich from the resource. Setting this option to true allows end users to control what to do if there was an exception in the aggregate method. For example to suppress the exception or set a custom message body etc.
<code>shareUnitOfWork</code>	<code>false</code>	Camel 2.16: Shares the unit of work with the parent and the resource exchange.

Enrich will by default not share unit of work between the parent exchange and the resource exchange. This means the resource exchange has its own individual unit of work. See Splitter for more information and example.

cachesize	Camel 2.16: Allows to configure the cache size for the ProducerCache which caches producers for reuse in the enrich. Will by default use the default cache size which is 1000. Setting the value to -1 allows to turn off the cache all together.
ignoreInvalidEndpoint	Camel 2.16: Whether to ignore an endpoint URI that could not be resolved. If disabled, Camel will throw an exception identifying the invalid endpoint URI.

Using the Fluent Builders

```
AggregationStrategy aggregationStrategy = ... from("direct:start").enrich("direct:resource", aggregationStrategy)
.to("direct:result"); from("direct:resource") ...
```

The content enricher (`enrich`) retrieves additional data from a *resource endpoint* in order to enrich an incoming message (contained in the *original exchange*). An aggregation strategy is used to combine the original exchange and the *resource exchange*. The first parameter of the `AggregationStrategy.aggregate(Exchange, Exchange)` method corresponds to the the original exchange, the second parameter the resource exchange. The results from the resource endpoint are stored in the resource exchange's out-message. Here's an example template for implementing an aggregation strategy:

```
java public class ExampleAggregationStrategy implements AggregationStrategy { public Exchange aggregate(Exchange original,
Exchange resource) { Object originalBody = original.getIn().getBody(); Object resourceResponse = resource.getIn().getBody();
Object mergeResult = ... // combine original body and resource response if (original.getPattern().isOutCapable()) {
original.getOut().setBody(mergeResult); } else { original.getIn().setBody(mergeResult); } return original; } }
```

Using this template the original exchange can be of any pattern. The resource exchange created by the enricher is always an in-out exchange.

Using Spring XML

The same example in the Spring DSL (Camel 2.15 or older)

```
xml<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring"> <route> <from uri="direct:start"/> <enrich
uri="direct:resource" strategyRef="aggregationStrategy"/> <to uri="direct:result"/> </route> <route> <from uri="direct:resource"/>
... </route> </camelContext> <bean id="aggregationStrategy" class="..." />
```

The same example in the Spring DSL (Camel 2.16 or newer)

```
xml<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring"> <route> <from uri="direct:start"/> <enrich
strategyRef="aggregationStrategy"/> <constant>direct:resource</constant> </enrich> <to uri="direct:result"/> </route> <route>
<from uri="direct:resource"/> ... </route> </camelContext> <bean id="aggregationStrategy" class="..." />
```

Aggregation strategy is optional

The aggregation strategy is optional. If you do not provide it Camel will by default just use the body obtained from the resource. `from("direct:start").enrich("direct:resource").to("direct:result")`;

In the route above the message sent to the `direct:result` endpoint will contain the output from the `direct:resource` as we do not use any custom aggregation.

And for Spring DSL (Camel 2.15 or older) just omit the `strategyRef` attribute:

```
xml <route> <from uri="direct:start"/> <enrich uri="direct:resource"/> <to uri="direct:result"/> </route>
```

And for Spring DSL (Camel 2.16 or newer) just omit the `strategyRef` attribute:

```
xml <route> <from uri="direct:start"/> <enrich> <constant>direct:resource</constant> </enrich> <to uri="direct:result"/> </route>
```

Using dynamic uris

Available as of Camel 2.16

From Camel 2.16 onwards enrich and pollEnrich supports using dynamic uris computed based on information from the current Exchange. For example to enrich from a HTTP endpoint where the header with key `orderId` is used as part of the content-path of the HTTP url:

```
from("direct:start").enrich().simple("http:myserver/${header.orderId}/order").to("direct:result");
```

And in XML DSL

```
xml <route> <from uri="direct:start"/> <enrich> <simple>http:myserver/${header.orderId}/order</simple> </enrich> <to
uri="direct:result"/> </route>
```

Content enrichment using pollEnrich

The `pollEnrich` works just as the `enrich` however as it uses a Polling Consumer we have 3 methods when polling

- receive
- receiveNoWait
- receive(timeout)

PollEnrich Options

confluenceTableSmall

Name	Default	Description
------	---------	-------------

	Value	
uri	The endpoint uri for the external service to enrich from. You must use either uri or ref. Important: From Camel 2.16 onwards, this option is removed, and you use an Expression to configure the uri, such as Simple or Constant or any other dynamic language that can compute the uri dynamically using values from the current Exchange.	
ref	Refers to the endpoint for the external service to enrich from. You must use either uri or ref. Important: From Camel 2.16 onwards, this option is removed, and you use an Expression to configure the uri, such as Simple or Constant or any other dynamic language that can compute the uri dynamically using values from the current Exchange.	
expression	Camel 2.16: Mandatory. The Expression to configure the uri, such as Simple or Constant or any other dynamic language that can compute the uri dynamically using values from the current Exchange.	
strategyRef	Refers to an AggregationStrategy to be used to merge the reply from the external service, into a single outgoing message. By default Camel will use the reply from the external service as outgoing message. From Camel 2.12 onwards you can also use a POJO as the AggregationStrategy, see the Aggregate page for more details.	
strategyMethodName	Camel 2.12: This option can be used to explicitly declare the method name to use, when using POJOs as the AggregationStrategy. See the Aggregate page for more details.	
strategyMethodAllowNull	false	Camel 2.12: If this option is false then the aggregate method is not used if there was no data to enrich. If this option is true then null values are used as the oldExchange (when no data to enrich), when using POJOs as the AggregationStrategy. See the Aggregate page for more details.
timeout	-1	Timeout in millis when polling from the external service. See below for important details about the timeout.
aggregateOnException	false	Camel 2.14: If this option is false then the aggregate method is not used if there was an exception thrown while trying to retrieve the data to enrich from the resource. Setting this option to true allows end users to control what to do if there was an exception in the aggregate method. For example to suppress the exception or set a custom message body etc.
cacheSize		Camel 2.16: Allows to configure the cache size for the ConsumerCache which caches consumers for reuse in the pollEnrich. Will by default use the default cache size which is 1000. Setting the value to -1 allows to turn off the cache all together.
ignoreInvalidEndpoint	false	Camel 2.16: Whether to ignore an endpoint URI that could not be resolved. If disabled, Camel will throw an exception identifying the invalid endpoint URI.

Good practice to use timeout value

By default Camel will use the receive. Which may block until there is a message available. It is therefore recommended to always provide a timeout value, to make this clear that we may wait for a message, until the timeout is hit.
If there is no data then the newExchange in the aggregation strategy is null.

You can pass in a timeout value that determines which method to use

- if timeout is -1 or other negative number then receive is selected (**Important:** the receive method may block if there is no message)
- if timeout is 0 then receiveNowait is selected
- otherwise receive(timeout) is selected

The timeout values is in millis.

Camel 2.15 or older - Data from current Exchange not used

pollEnrich does not access any data from the current Exchange which means when polling it cannot use any of the existing headers you may have set on the Exchange. For example you cannot set a filename in the Exchange .FILE_NAME header and use pollEnrich to consume only that file. For that you must set the filename in the endpoint URI.

From **Camel 2.16** onwards both enrich and pollEnrich supports dynamic endpoints that uses an Expression to compute the uri, which allows to use data from the current Exchange. In other words all what is told above no longer apply and it just works.

Example

In this example we enrich the message by loading the content from the file named inbox/data.txt.

```
from("direct:start").pollEnrich("file:inbox?fileName=data.txt").to("direct:result");
```

And in XML DSL (Camel 2.15 or older) you do:

```
xml <route> <from uri="direct:start"/> <pollEnrich uri="file:inbox?fileName=data.txt"/> <to uri="direct:result"/> </route>
```

And in XML DSL (Camel 2.16 or newer) you do:

```
xml <route> <from uri="direct:start"/> <pollEnrich> <constant>file:inbox?fileName=data.txt</constant> </pollEnrich> <to uri="direct:result"/> </route>
```

If there is no file then the message is empty. We can use a timeout to either wait (potentially forever) until a file exists, or use a timeout to wait a certain period.

For example to wait up to 5 seconds you can do (Camel 2.15 or older):

```
xml <route> <from uri="direct:start"/> <pollEnrich uri="file:inbox?fileName=data.txt" timeout="5000"/> <to uri="direct:result"/> </route>
```

For example to wait up to 5 seconds you can do (Camel 2.16 or newer):

```
xml <route> <from uri="direct:start"/> <pollEnrich timeout="5000"> <constant>file:inbox?fileName=data.txt</constant> </pollEnrich> <to uri="direct:result"/> </route>
```

Using dynamic uris

Available as of Camel 2.16

From Camel 2.16 onwards enrich and pollEnrich supports using dynamic uris computed based on information from the current Exchange. For example to pollEnrich from an endpoint that uses a header to indicate a SEDA queue name:

```
from("direct:start") .pollEnrich().simple("seda:${header.name}") .to("direct:result");
```

And in XML DSL

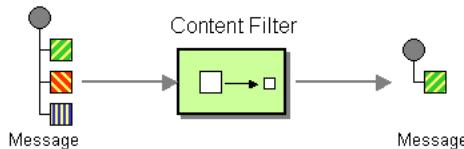
```
xml <route> <from uri="direct:start"/> <pollEnrich> <simple>seda:${header.name}</simple> </pollEnrich> <to uri="direct:result"/> </route>
```

Using This Pattern

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>
```

You can also use XPath to filter out part of the message you are interested in:

```
<route>
  <from uri="activemq:Input"/>
  <setBody><xpath resultType="org.w3c.dom.Document">/foo:bar</xpath></setBody>
  <to uri="activemq:Output"/>
</route>
```

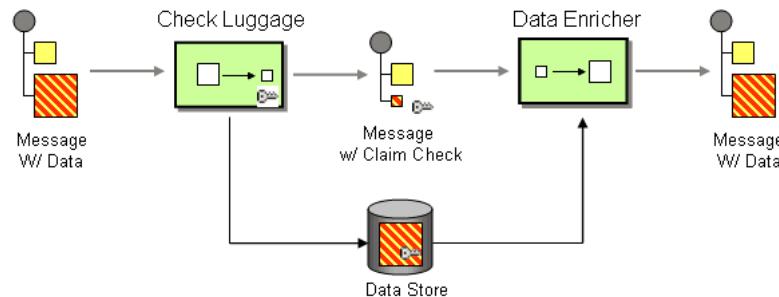
Using This Pattern

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.

Claim Check

The Claim Check from the EIP patterns allows you to replace message content with a claim check (a unique key), which can be used to retrieve the message content at a later time. The message content is stored temporarily in a persistent store like a database or file system. This pattern is very useful when message content is very large (thus it would be expensive to send around) and not all components require all information.

It can also be useful in situations where you cannot trust the information with an outside party; in this case, you can use the Claim Check to hide the sensitive portions of data.



Example

In this example we want to replace a message body with a claim check, and restore the body at a later step.

Using the Fluent Builders

```
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/ClaimCheckTest.java}
```

Using the Spring XML Extensions

```
xml <route> <from uri="direct:start"/> <pipeline> <to uri="bean:checkLuggage"/> <to uri="mock:testCheckpoint"/> <to uri="bean:dataEnricher"/> <to uri="mock:result"/> </pipeline> </route>
```

The example route is pretty simple - its just a Pipeline. In a real application you would have some other steps where the `mock:testCheckpoint` endpoint is in the example.

The message is first sent to the `checkLuggage` bean which looks like
`{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/ClaimCheckTest.java}`
This bean stores the message body into the data store, using the `custId` as the claim check. In this example, we're just using a `HashMap` to store the message body; in a real application you would use a database or file system, etc. Next the claim check is added as a message header for use later. Finally we remove the body from the message and pass it down the pipeline.

The next step in the pipeline is the `mock:testCheckpoint` endpoint which is just used to check that the message body is removed, claim check added, etc.

To add the message body back into the message, we use the `dataEnricher` bean which looks like
`{snippet:id=e3|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/ClaimCheckTest.java}`
This bean queries the data store using the claim check as the key and then adds the data back into the message. The message body is then removed from the data store and finally the claim check is removed. Now the message is back to what we started with!

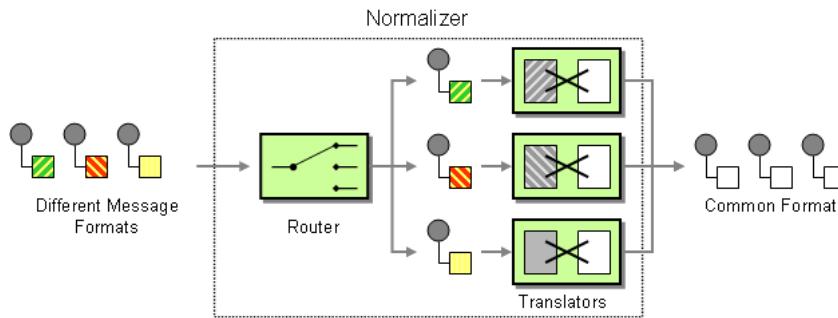
For full details, check the example source here:

[camel-core/src/test/java/org/apache/camel/processor/ClaimCheckTest.java](#)

Using This Pattern

Normalizer

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



Example

This example shows a Message Normalizer that converts two types of XML messages into a common format. Messages in this common format are then filtered.

Using the Fluent Builders

```
{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/NormalizerTest.java}
In this case we're using a Java bean as the normalizer. The class looks like this
```

```
{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/MyNormalizer.java}
```

Using the Spring XML Extensions

The same example in the Spring DSL

```
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/normalizer.xml}
```

See Also

- Message Router
- Content Based Router
- Message Translator

Using This Pattern

SORT

Sort can be used to sort a message. Imagine you consume text files and before processing each file you want to be sure the content is sorted.

Sort will by default sort the body using a default comparator that handles numeric values or uses the string representation. You can provide your own comparator, and even an expression to return the value to be sorted. Sort requires the value returned from the expression evaluation is convertible to `java.util.List` as this is required by the JDK sort operation.

Options

Name	Default Value	Description
<code>comparatorRef</code>		Refers to a custom <code>java.util.Comparator</code> to use for sorting the message body. Camel will by default use a comparator which does a A_Z sorting.

Using from Java DSL

In the route below it will read the file content and tokenize by line breaks so each line can be sorted.

```
from("file://inbox").sort(body()).tokenize("\n").to("bean:MyServiceBean.processLine");
```

You can pass in your own comparator as a 2nd argument:

```
from("file://inbox").sort(body().tokenize("\n"), new MyReverseComparator()).to("bean:MyServiceBean.processLine");
```

Using from Spring DSL

In the route below it will read the file content and tokenize by line breaks so each line can be sorted.

```
Listing 1. Camel 2.7 or better
<route>
  <from uri="file://inbox"/>
```

```
<sort>
  <simple>body</simple>
</sort>
<beanRef ref="myServiceBean" method="processLine"/>
</route>
```

Listing 1. Camel 2.6 or older

```
<route>
  <from uri="file://inbox"/>
  <sort>
    <expression>
      <simple>body</simple>
    </expression>
  </sort>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>
```

And to use our own comparator we can refer to it as a spring bean:

Listing 1. Camel 2.7 or better

```
<route>
  <from uri="file://inbox"/>
  <sort comparatorRef="myReverseComparator">
    <simple>body</simple>
  </sort>
  <beanRef ref="MyServiceBean" method="processLine"/>
</route>

<bean id="myReverseComparator" class="com.mycompany.MyReverseComparator"/>
```

Listing 1. Camel 2.6 or older

```
<route>
  <from uri="file://inbox"/>
  <sort comparatorRef="myReverseComparator">
    <expression>
      <simple>body</simple>
    </expression>
  </sort>
  <beanRef ref="MyServiceBean" method="processLine"/>
</route>

<bean id="myReverseComparator" class="com.mycompany.MyReverseComparator"/>
```

Besides `<simple>`, you can supply an expression using any language you like, so long as it returns a list.

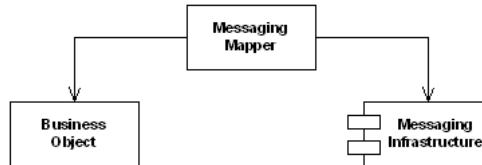
Using This Pattern

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.



Example

The following example demonstrates the use of a Bean component to map between two messaging system

Using the Fluent Builders

```
from("activemq:foo")
  .beanRef("transformerBean", "transform")
  .to("jms:bar");
```

Using the Spring XML Extensions

```
<route>
  <from uri="activemq:foo"/>
  <bean ref="transformerBean" method="transform" />
  <to uri="jms:bar"/>
</route>
```

See also

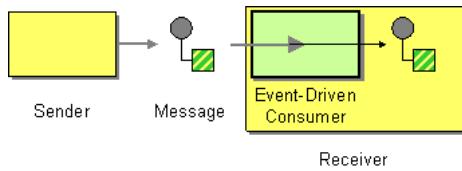
- Message Translator
- Type Converter
- CXF for JAX-WS support for binding business logic to messaging & web services
- Pojo
- Bean

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.

Example

The following demonstrates a Processor defined in the Camel Registry which is invoked when an event occurs from a JMS queue

Using the Fluent Builders

```
from("jms:queue:foo")
  .processRef("processor");
```

Using the Spring XML Extensions

```
<route>
  <from uri="jms:queue:foo"/>
  <to uri="processor"/>
</route>
```

For more details see

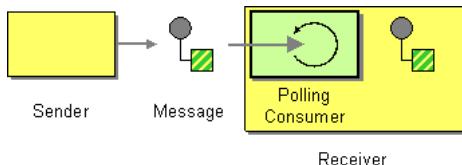
- Message
- Message Endpoint

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.



In Java:

```
javaEndpoint endpoint = context.getEndpoint("activemq:my.queue"); PollingConsumer consumer =
endpoint.createPollingConsumer(); Exchange exchange = consumer.receive();
The consumerTemplate (discussed below) is also available.
```

There are three 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

EventDrivenPollingConsumer Options

The EventDrivenPollingConsumer (the default implementation) supports the following options:
confluenceTableSmall

Option	Default	Description
pollingConsumerQueueSize	1000	Camel 2.14/2.13.1/2.12.4: The queue size for the internal hand-off queue between the polling consumer, and producers sending data into the queue.
pollingConsumerBlockWhenFull	true	Camel 2.14/2.13.1/2.12/4: Whether to block any producer if the internal queue is full.
pollingConsumerBlockTimeout	0	Camel 2.16: To use a timeout (in milliseconds) when the producer is blocked if the internal queue is full. If the value is 0 or negative then no timeout is in use. If a timeout is triggered then a ExchangeTimedoutException is thrown.

Notice that some Camel Components has their own implementation of PollingConsumer and therefore do not support the options above.

You can configure these options in endpoints URIs, such as shown below:

```
javaEndpoint endpoint = context.getEndpoint("file:inbox?pollingConsumerQueueSize=50"); PollingConsumer consumer =
endpoint.createPollingConsumer(); Exchange exchange = consumer.receive(5000);
```

ConsumerTemplate

The consumerTemplate is a template much like Spring's JmsTemplate or JdbcTemplate supporting the Polling Consumer EIP. With the template you can consume Exchanges from an Endpoint. The template supports the three operations listed above. However, it also includes convenient methods for returning the body, etc consumeBody.

Example:

```
Exchange exchange = consumerTemplate.receive("activemq:my.queue");
```

Or to extract and get the body you can do:

```
Object body = consumerTemplate.receiveBody("activemq:my.queue");
```

And you can provide the body type as a parameter and have it returned as the type:

```
String body = consumerTemplate.receiveBody("activemq:my.queue", String.class);
```

You get hold of a consumerTemplate from the CamelContext with the createConsumerTemplate operation:

```
ConsumerTemplate consumer = context.createConsumerTemplate();
```

Using ConsumerTemplate with Spring DSL

With the Spring DSL we can declare the consumer in the CamelContext with the consumerTemplate tag, just like the ProducerTemplate. The example below illustrates this:{snippet:id=e1|lang=xml|url=camel/components/camel-spring/src/test/resources/org/apache/camel/spring/SpringConsumerTemplateTest-context.xml} Then we can get leverage Spring to inject the ConsumerTemplate in our java class. The code below is part of an unit test but it shows how the consumer and producer can work together.{snippet:id=e1|lang=java|url=camel/components/camel-spring/src/test/java/org/apache/camel/spring/SpringConsumerTemplateTest.java}

Timer Based Polling Consumer

In this sample we use a Timer to schedule a route to be started every 5th second and invoke our bean `MycoolBean` where we implement the business logic for the Polling Consumer. Here we want to consume all messages from a JMS queue, process the message and send them to the next queue.

First we setup our route as:{snippet:id=e1|lang=java|url=camel/tags/camel-2.6.0/components/camel-jms/src/test/java/org/apache/camel/component/jms/JmsTimerBasedPollingConsumerTest.java}And then we have out logic in our bean:{snippet:id=e2|lang=java|url=camel/tags/camel-2.6.0/components/camel-jms/src/test/java/org/apache/camel/component/jms/JmsTimerBasedPollingConsumerTest.java}

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 is 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](#)
 - [Scheduler](#)
 - [Atom](#)
 - [Beanstalk](#)
 - [File](#)
 - [FTP](#)
 - [hbase](#)
 - [iBATIS](#)
 - [JPA](#)
 - [Mail](#)
 - [MyBatis](#)
 - [Quartz](#)
 - [SNMP](#)
 - [AWS-S3](#)
 - [AWS-SQS](#)

ScheduledPollConsumer Options

The `ScheduledPollConsumer` supports the following options:

confluenceTableSmall

Option	Default	Description
<code>backoffErrorThreshold</code>	0	Camel 2.12: The number of subsequent error polls (failed due some error) that should happen before the <code>backoffMultiplier</code> should kick-in.
<code>backoffIdleThreshold</code>	0	Camel 2.12: The number of subsequent idle polls that should happen before the <code>backoffMultiplier</code> should kick-in.
<code>backoffMultiplier</code>	0	Camel 2.12: To let the scheduled polling consumer back-off if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then <code>backoffIdleThreshold</code> and/or <code>backoffErrorThreshold</code> must also be configured.
<code>delay</code>	500	Milliseconds before the next poll.
<code>greedy</code>	false	Camel 2.10.6/2.11.1: If greedy is enabled, then the <code>ScheduledPollConsumer</code> will run immediately again, if the previous run polled 1 or more messages.
<code>initialDelay</code>	1000	Milliseconds before the first poll starts.
<code>pollStrategy</code>		A pluggable <code>org.apache.camel.pollingConsumerPollingStrategy</code> allowing you to provide your custom implementation to control error handling usually occurred during the <code>poll</code> operation before an Exchange has been created and routed in Camel. In other words the error occurred while the polling was gathering information, for instance access to a file network failed so Camel cannot access it to scan for files.

		The default implementation will log the caused exception at WARN level and ignore it.
runLoggingLevel	TRACE	Camel 2.8: The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.
scheduledExecutorService	null	Camel 2.10: Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool. This option allows you to share a thread pool among multiple consumers.
scheduler	null	Camel 2.12: Allow to plugin a custom org.apache.camel.spi.ScheduledPollConsumerScheduler to use as the scheduler for firing when the polling consumer runs. The default implementation uses the ScheduledExecutorService and there is a Quartz2, and Spring based which supports CRON expressions. Notice: If using a custom scheduler then the options for initialDelay, useFixedDelay, timeUnit and scheduledExecutorService may not be in use. Use the text quartz2 to refer to use the Quartz2 scheduler; and use the text spring to use the Spring based; and use the text #myscheduler to refer to a custom scheduler by its id in the Registry. See Quartz2 page for an example.
scheduler.xxx	null	Camel 2.12: To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.
sendEmptyMessageWhenIdle	false	Camel 2.9: If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.
startScheduler	true	Whether the scheduler should be auto started.
timeUnit	TimeUnit.MILLISECONDS	Time unit for initialDelay and delay options.
useFixedDelay		Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details. In Camel 2.7.x or older the default value is false. From Camel 2.8: the default value is true.

Using backoff to Let the Scheduler be Less Aggressive

Available as of Camel 2.12

The scheduled Polling Consumer is by default static by using the same poll frequency whether or not there is messages to pickup or not.

From **Camel 2.12:** you can configure the scheduled Polling Consumer to be more dynamic by using backoff. This allows the scheduler to skip N number of polls when it becomes idle, or there has been X number of errors in a row. See more details in the table above for the backoffxxx options.

For example to let a FTP consumer back-off if its becoming idle for a while you can do:

```
javafrom("ftp://myserver?username=foo&password=secret&delete=true&delay=5s&backoffMultiplier=6&backoffIdleThreshold=5")
.to("bean:processFile");
```

In this example, the FTP consumer will poll for new FTP files every 5th second. But if it has been idle for 5 attempts in a row, then it will back-off using a multiplier of 6, which means it will now poll every $5 \times 6 = 30$ second instead. When the consumer eventually pickup a file, then the back-off will reset, and the consumer will go back and poll every 5th second again.

Camel will log at DEBUG level using org.apache.camel.impl.ScheduledPollConsumer when back-off is kicking-in.

About Error Handling and Scheduled Polling Consumers

ScheduledPollConsumer is scheduled based and its run method is invoked periodically based on schedule settings. But errors can also occur when a poll is being executed. For instance if Camel should poll a file network, and this network resource is not available then a java.io.IOException could occur. As this error happens **before** any Exchange has been created and prepared for routing, then the regular Error handling in Camel does not apply. So what does the consumer do then? Well the exception is propagated back to the run method where its handled. Camel will by default log the exception at WARN level and then ignore it. At next schedule the error could have been resolved and thus being able to poll the endpoint successfully.

Using a Custom Scheduler

Available as of Camel 2.12:

The SPI interface org.apache.camel.spi.ScheduledPollConsumerScheduler allows to implement a custom scheduler to control when the Polling Consumer runs. The default implementation is based on the JDks ScheduledExecutorService with a single thread in the thread pool. There is a CRON based implementation in the Quartz2, and Spring components.

For an example of developing and using a custom scheduler, see the unit test `org.apache.camel.component.file.FileConsumerCustomSchedulerTest` from the source code in `camel-core`.

Error Handling When Using PollingConsumerPollStrategy

`org.apache.camel.PollingConsumerPollStrategy` is a pluggable strategy that you can configure on the `ScheduledPollConsumer`. The default implementation `org.apache.camel.impl.DefaultPollingConsumerPollStrategy` will log the caused exception at `WARN` level and then ignore this issue.

The strategy interface provides the following three methods:

- `begin`
 - `void begin(Consumer consumer, Endpoint endpoint)`
- `begin (Camel 2.3)`
 - `boolean begin(Consumer consumer, Endpoint endpoint)`
- `commit`
 - `void commit(Consumer consumer, Endpoint endpoint)`
- `commit (Camel 2.6)`
 - `void commit(Consumer consumer, Endpoint endpoint, int polledMessages)`
- `rollback`
 - `boolean rollback(Consumer consumer, Endpoint endpoint, int retryCounter, Exception e) throws Exception`

In **Camel 2.3**: the `begin` method returns a `boolean` which indicates whether or not to skipping polling. So you can implement your custom logic and return `false` if you do not want to poll this time.

In **Camel 2.6**: the `commit` method has an additional parameter containing the number of message that was actually polled. For example if there was no messages polled, the value would be zero, and you can react accordingly.

The most interesting is the `rollback` as it allows you do handle the caused exception and decide what to do.

For instance if we want to provide a retry feature to a scheduled consumer we can implement the `PollingConsumerPollStrategy` method and put the retry logic in the `rollback` method. Lets just retry up till three times:
`javapublic boolean rollback(Consumer consumer, Endpoint endpoint, int retryCounter, Exception e) throws Exception { if (retryCounter < 3) { // return true to tell Camel that it should retry the poll immediately return true; } // okay we give up do not retry anymore return false; }`

Notice that we are given the `consumer` as a parameter. We could use this to `restart` the consumer as we can invoke `stop` and `start`:

`java// error occurred lets restart the consumer, that could maybe resolve the issue consumer.stop(); consumer.start();`

Note: if you implement the `begin` operation make sure to avoid throwing exceptions as in such a case the `poll` operation is not invoked and Camel will invoke the `rollback` directly.

Configuring an Endpoint to Use PollingConsumerPollStrategy

To configure an Endpoint to use a custom `PollingConsumerPollStrategy` you use the option `pollStrategy`. For example in the file consumer below we want to use our custom strategy defined in the Registry with the bean id `myPoll`:

```
from("file:/inbox/?pollStrategy=#myPoll") .to("activemq:queue:inbox")
```

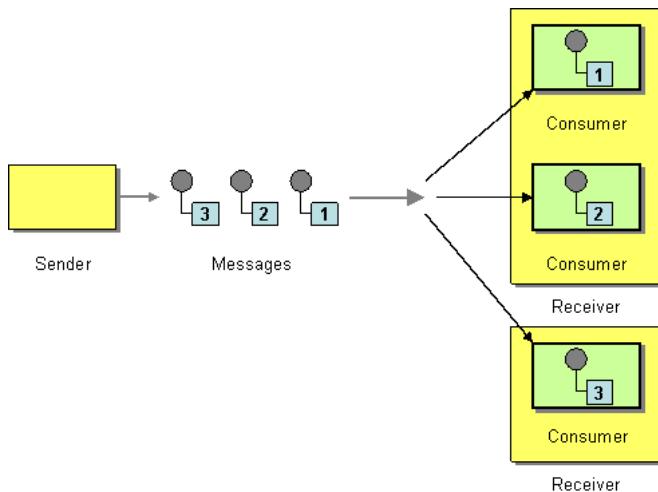
Using This Pattern

See Also

- POJO Consuming
- Batch Consumer

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:-

- Seda for SEDA based concurrent processing using a thread pool
- JMS for distributed SEDA based concurrent processing with queues which support reliable load balancing, 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 in Spring DSL

```
<route>
  <from uri="jms:MyQueue?concurrentConsumers=5"/>
  <to uri="bean:someBean"/>
</route>
```

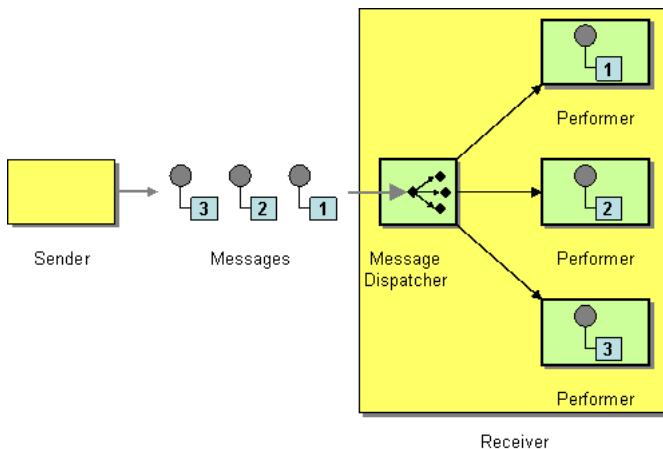
Or just run multiple JVMs of any ActiveMQ or JMS route 😊

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.

Example

The following example demonstrates Message Dispatcher pattern using the Competing Consumers functionality of the JMS component to offload messages to a Content Based Router and custom Processors registered in the Camel Registry running in separate threads from originating consumer.

Using the Fluent Builders

```
from("jms:queue:foo?concurrentConsumers=5")
    .threads(5)
    .choice()
        .when(header("type").isEqualTo("A"))
            .processRef("messageDispatchProcessorA")
        .when(header("type").isEqualTo("B"))
            .processRef("messageDispatchProcessorB")
        .when(header("type").isEqualTo("C"))
            .processRef("messageDispatchProcessorC")
        .otherwise()
            .to("jms:queue:invalidMessageType");
```

Using the Spring XML Extensions

```
<route>
    <from uri="jms:queue:foo?concurrentConsumers=5"/>
    <threads poolSize="5">
        <choice>
            <when>
                <simple>${in.header.type} == 'A'</simple>
                <to ref="messageDispatchProcessorA"/>
            </when>
            <when>
                <simple>${in.header.type} == 'B'</simple>
                <to ref="messageDispatchProcessorB"/>
            </when>
            <when>
                <simple>${in.header.type} == 'C'</simple>
                <to ref="messageDispatchProcessorC"/>
            </when>
            <otherwise>
                <to uri="jms:queue:invalidMessageType"/>
            </otherwise>
        </choice>
    </threads>
</route>
```

See Also

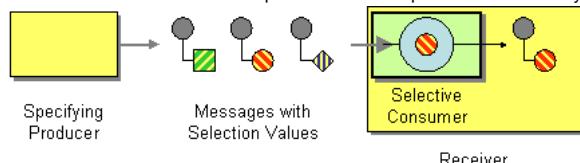
- JMS
- Selective Consumer
- Content Based Router
- Endpoint

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() {
        errorHandler(deadLetterChannel("mock:error"));

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

Using the Spring XML Extensions

```

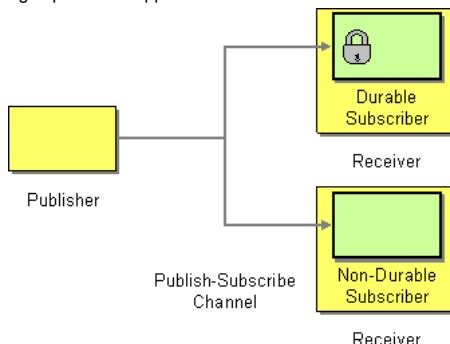
<bean id="myProcessor" class="org.apache.camel.builder.MyProcessor"/>
<camelContext errorHandlerRef="errorHandler" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:a"/>
        <filter>
            <xpath>$foo = 'bar'</xpath>
            <process ref="myProcessor"/>
        </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 Seda for non-durable.

Here is a simple example of creating durable subscribers to a JMS topic

Using the Fluent Builders

```
from("direct:start").to("activemq:topic:foo");
from("activemq:topic:foo?clientId=1&durableSubscriptionName=bar1").to("mock:result1");
from("activemq:topic:foo?clientId=2&durableSubscriptionName=bar2").to("mock:result2");
```

Using the Spring XML Extensions

```
<route>
  <from uri="direct:start"/>
  <to uri="activemq:topic:foo"/>
</route>

<route>
  <from uri="activemq:topic:foo?clientId=1&durableSubscriptionName=bar1"/>
  <to uri="mock:result1"/>
</route>

<route>
  <from uri="activemq:topic:foo?clientId=2&durableSubscriptionName=bar2"/>
  <to uri="mock:result2"/>
</route>
```

Here is another example of JMS durable subscribers, but this time using virtual topics (recommended by AMQ over durable subscriptions)

Using the Fluent Builders

```
from("direct:start").to("activemq:topic:VirtualTopic.foo");
from("activemq:queue:Consumer.1.VirtualTopic.foo").to("mock:result1");
from("activemq:queue:Consumer.2.VirtualTopic.foo").to("mock:result2");
```

Using the Spring XML Extensions

```
<route>
  <from uri="direct:start"/>
  <to uri="activemq:topic:VirtualTopic.foo"/>
</route>

<route>
  <from uri="activemq:queue:Consumer.1.VirtualTopic.foo"/>
  <to uri="mock:result1"/>
</route>

<route>
  <from uri="activemq:queue:Consumer.2.VirtualTopic.foo"/>
  <to uri="mock:result2"/>
</route>
```

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 `IdempotentRepository` 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.

Camel will add the message id eagerly to the repository to detect duplication also for Exchanges currently in progress. On completion Camel will remove the message id from the repository if the Exchange failed, otherwise it stays there.

Camel provides the following Idempotent Consumer implementations:

- MemoryIdempotentRepository
- FileIdempotentRepository
- HazelcastIdempotentRepository (**Available as of Camel 2.8**)
- JdbcMessageIdRepository (**Available as of Camel 2.7**)
- JpaMessageIdRepository
- InfinispanIdempotentRepository (**Available as of Camel 2.13.0**)
- JCachecIdempotentRepository (**Available as of Camel 2.17.0**)
- SpringCachecIdempotentRepository (**Available as of Camel 2.17.1**)
- EhcacheIdempotentRepository (**Available as of Camel 2.18.0**)
- KafkaIdempotentRepository (**Available as of Camel 2.19.0**)

Options

The Idempotent Consumer has the following options:

Option	Default	Description
eager	true	Eager controls whether Camel adds the message to the repository before or after the exchange has been processed. If enabled before then Camel will be able to detect duplicate messages even when messages are currently in progress. By disabling Camel will only detect duplicates when a message has successfully been processed.
messageIdRepositoryRef	null	A reference to a <code>IdempotentRepository</code> to lookup in the registry. This option is mandatory when using XML DSL.
skipDuplicate	true	Camel 2.8: Sets whether to skip duplicate messages. If set to <code>false</code> then the message will be continued. However the Exchange has been marked as a duplicate by having the <code>Exchange.DUPLICATE_MESSAGE</code> exchange property set to a <code>Boolean.TRUE</code> value.
removeOnFailure	true	Camel 2.9: Sets whether to remove the id of an Exchange that failed. Camel 2.16: Sets whether to complete the idempotent consumer eager or when the exchange is done. If this option is true to complete eager, then the idempotent consumer will trigger its completion when the exchange reached the end of the block of the idempotent consumer pattern. So if the exchange is continued routed after the block ends, then whatever happens there does not affect the state.
completionEager	false	If this option is false (default) to not complete eager, then the idempotent consumer will complete when the exchange is done being routed. So if the exchange is continued routed after the block ends, then whatever happens there also affect the state. For example if the exchange failed due to an exception, then the state of the idempotent consumer will be a rollback.

Using the Fluent Builders

The following example will use the header `myMessageId` to filter out duplicates{snippet:id=idempotent|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/builder/RouteBuilderTest.java}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{snippet:id=idempotent|lang=java|url=camel/trunk/components/camel-jpa/src/test/java/org/apache/camel/processor/jpa/JpaIdempotentConsumerTest.java}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.

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

Spring XML example

The following example will use the header `myMessageId` to filter out duplicates{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringIdempotentConsumerTest.xml}

How to handle duplicate messages in the route

Available as of Camel 2.8

You can now set the `skipDuplicate` option to `false` which instructs the idempotent consumer to route duplicate messages as well. However the duplicate message has been marked as duplicate by having a property on the Exchange set to `true`. We can leverage this fact by using a Content Based Router or Message Filter to detect this and handle duplicate messages.

For example in the following example we use the Message Filter to send the message to a duplicate endpoint, and then stop continue routing that message.
 {snippet:id=e1|lang=java|title=Filter duplicate messages|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/IdempotentConsumerTest.java}
 The sample example in XML DSL would be:
 {snippet:id=e1|lang=xml|title=Filter duplicate messages|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringIdempotentConsumerNoSkipDuplicateFilterTest.xml}

How to handle duplicate message in a clustered environment with a data grid

Available as of Camel 2.8

If you have running Camel in a clustered environment, a in memory idempotent repository doesn't work (see above). You can setup either a central database or use the idempotent consumer implementation based on the Hazelcast data grid. Hazelcast finds the nodes over multicast (which is default - configure Hazelcast for tcp-ip) and creates automatically a map based repository:

```
java HazelcastIdempotentRepository idempotentRepo = new HazelcastIdempotentRepository("myrepo");
from("direct:in").idempotentConsumer(header("messageId"), idempotentRepo).to("mock:out");
You have to define how long the repository should hold each message id (default is to delete it never). To avoid that you run out of memory you should create an eviction strategy based on the Hazelcast configuration. For additional information see camel-hazelcast.
```

See this little tutorial, how setup such an idempotent repository on two cluster nodes using Apache Karaf.

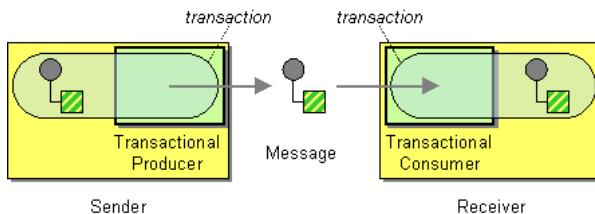
Available as of Camel 2.13.0

Another option for using Idempotent Consumer in a clustered environment is Infinispan. Infinispan is a data grid with replication and distribution clustering support. For additional information see [camel-infinispan](#).

Using This Pattern

Transactional Client

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



Transaction Oriented Endpoints 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.

Configuration of Redelivery

The redelivery in transacted mode is **not** handled by Camel but by the backing system (the transaction manager). In such cases you should resort to the backing system how to configure the redelivery.

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.

```
xml <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.

```
java 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 under the covers for declaring the transaction demarcation to use. So you will need to add something like the following to your spring xml:

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

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

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

```
javapublic void configure() { ... Policy required = bean(SpringTransactionPolicy.class, "PROPAGATION_REQUIRED"); Policy
requirenew = bean(SpringTransactionPolicy.class, "PROPAGATION_REQUIRED"); ... }
```

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

```
java // 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");
```

OSGi Blueprint

If you are using OSGi Blueprint then you most likely have to explicit declare a policy and refer to the policy from the transacted in the route.

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

And then refer to "required" from the route:

```
xml<route> <from uri="activemq:queue:foo"/> <transacted ref="required"/> <to uri="activemq:queue:bar"/> </route>
```

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 DataSource TransactionManager that is doing the heavy lifting of ensuring our transactional policies. You are of course free to use any of the Spring based TransactionManager, eg. if you are in a full blown J2EE container you could use JTA or the WebLogic or WebSphere specific managers.

As we use the new convention over configuration we do **not** need to configure a transaction policy bean, so we do not have any PROPAGATION_REQUIRED beans. All the beans needed to be configured is **standard** Spring beans only, eg. there are no Camel specific configuration at all.{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/interceptor/springTransactionalClientDataSourceMinimalConfiguration.xml}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. Notice that we mark each route as transacted using the **transacted** tag.

```
{snippet:id=e2|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/interceptor/springTransactionalClientDataSourceMinimalConfiguration.xml}That is all that is needed to configure a Camel route as being transacted. Just remember to use the transacted DSL. The rest is standard Spring XML to setup the transaction manager.
```

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.

First we configure the standard Spring XML to declare a JMS connection factory, a JMS transaction manager and our ActiveMQ component that we use in our routing.{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-jms/src/test/resources/org/apache/camel/component/jms/tx/TransactionMinimalConfigurationTest.xml}And then we configure our routes. Notice that all we have to do is mark the route as transacted using the **transacted** tag.

```
{snippet:id=e2|lang=xml|url=camel/trunk/components/camel-jms/src/test/resources/org/apache/camel/component/jms/tx/TransactionMinimalConfigurationTest.xml}
```

Transaction error handler

When a route is marked as transacted using **transacted** Camel will automatic use the TransactionErrorHandler as Error Handler. It supports basically the same feature set as the DefaultErrorHandler, so you can for instance use Exception Clause as well.

Integration Testing with Spring

An Integration Test here means a test runner class annotated @RunWith(SpringJUnit4ClassRunner.class).

When following the Spring Transactions documentation it is tempting to annotate your integration test with @Transactional then seed your database before firing up the route to be tested and sending a message in. This is incorrect as Spring will have an in-progress transaction, and Camel will wait on this before proceeding, leading to the route timing out.

Instead, remove the @Transactional annotation from the test method and seed the test data within a TransactionTemplate execution which will ensure the data is committed to the database before Camel attempts to pick up and use the transaction manager. A simple example can be found on GitHub.

Spring's transactional model ensures each transaction is bound to one thread. A Camel route may invoke additional threads which is where the blockage may occur. This is not a fault of Camel but as the programmer you must be aware of the consequences of beginning a transaction in a test thread and expecting a separate thread created by your Camel route to be participate, which it cannot. You can, in your test, mock the parts that cause separate threads to avoid this issue.

USING MULTIPLE ROUTES WITH DIFFERENT PROPAGATION BEHAVIORS

Available as of Camel 2.2

Suppose you want to route a message through two routes and by which the 2nd route should run in its own transaction. How do you do that? You use propagation behaviors for that where you configure it as follows:

- The first route use PROPAGATION_REQUIRED
- The second route use PROPAGATION_REQUIRE_NEW

This is configured in the Spring XML file:{snippet:id=e1|lang=xmll|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/interceptor/MixedTransactionPropagationTest.xml}Then in the routes you use transacted DSL to indicate which of these two propagations it uses.{snippet:id=e1|lang=java|url=camel/trunk/components/camel-spring/src/test/java/org/apache/camel/spring/interceptor/MixedTransactionPropagationTest.java}Notice how we have configured the onException in the 2nd route to indicate in case of any exceptions we should handle it and just rollback this transaction. This is done using the markRollbackOnlyLast which tells Camel to only do it for the current transaction and not globally.

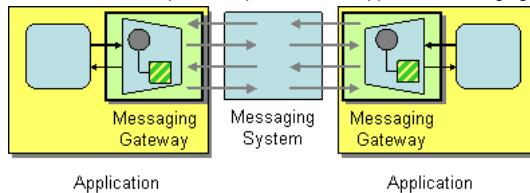
See Also

- Error handling in Camel
- TransactionErrorHandler
- Error Handler
- JMS

Using This Pattern

Messaging Gateway

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



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

However you may want to read the Using CamelProxy documentation as a true Messaging Gateway EIP solution.

Another approach is to use @Produce which you can read about in POJO Producing which also can be used as a Messaging Gateway EIP solution.

Example

The following example shows how the CXF and Bean components can be used to abstract the developer from the underlying messaging system API

Using the Fluent Builders

```
from("cxf:bean:soapMessageEndpoint")
.to("bean:testBean?method=processSOAP");
```

Using the Spring XML Extensions

```
<route>
    <from uri="cxf:bean:soapMessageEndpoint"/>
    <to uri="bean:testBean?method=processSOAP"/>
</route>
```

See Also

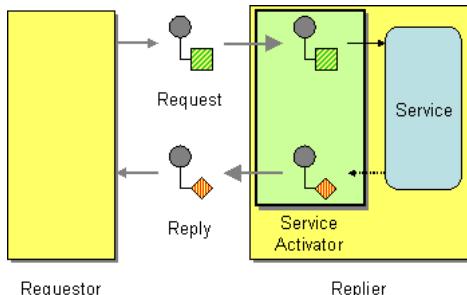
- Bean
- CXF
- Using CamelProxy
- POJO Producing
- Spring Remoting

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 way to bind the message exchange to a Java interface/service where the route defines the endpoints and wires it up to the bean.

In addition you can use the Bean Integration to wire messages to a bean using annotation.

Here is a simple example of using a Direct endpoint to create a messaging interface to a Pojo Bean service.

Using the Fluent Builders

```
from("direct:invokeMyService").to("bean:myService");
```

Using the Spring XML Extensions

```
<route>
    <from uri="direct:invokeMyService"/>
    <to uri="bean:myService"/>
</route>
```

See Also

- Bean
- Pojo
- CXF

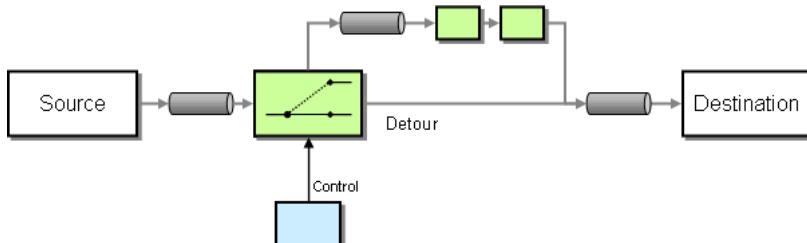
Using This Pattern

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

Detour

The Detour from the EIP patterns allows you to send messages through additional steps if a control condition is met. It can be useful for turning on extra validation, testing, debugging code when needed.



Example

In this example we essentially have a route like `from("direct:start").to("mock:result")` with a conditional detour to the `mock:detour` endpoint in the middle of the route..

```
from("direct:start").choice()
    .when().method("controlBean", "isDetour").to("mock:detour").end()
    .to("mock:result");
```

Using the Spring XML Extensions

```
<route>
    <from uri="direct:start"/>
    <choice>
        <when>
            <method bean="controlBean" method="isDetour"/>
        <to uri="mock:detour"/>
        </when>
    </choice>
    <to uri="mock:result"/>
</route>
```

whether the detour is turned on or off is decided by the `ControlBean`. So, when the detour is on the message is routed to `mock:detour` and then `mock:result`. When the detour is off, the message is routed to `mock:result`.

For full details, check the example source here:

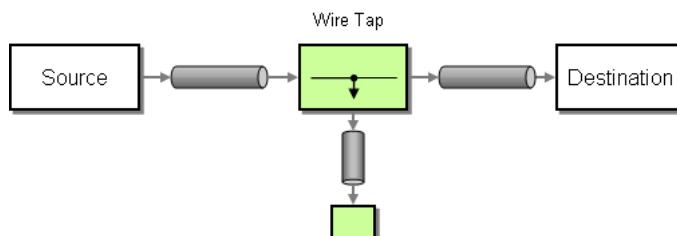
[camel-core/src/test/java/org/apache/camel/processor/DetourTest.java](#)

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.

Wire Tap

Wire Tap (from the EIP patterns) allows you to route messages to a separate location while they are being forwarded to the ultimate destination.



Streams

If you Wire Tap a stream message body then you should consider enabling Stream caching to ensure the message body can be read at each endpoint. See more details at [Stream caching](#).

Options

Name	Default	Description
uri		Mandatory: The URI of the endpoint to which the wire-tapped message should be sent. From Camel 2.16 : support for dynamic to URIs is as documented in Message Endpoint .
executorServiceRef		Reference ID of a custom Thread Pool to use when processing the wire-tapped messages. When not set, Camel will use an instance of the default thread pool.
processorRef		Reference ID of a custom Processor to use for creating a new message. See " Sending a New Exchange " below.
copy	true	Camel 2.3 : Whether to copy the Exchange before wire-tapping the message.
onPrepareRef		Camel 2.8 : Reference identifier of a custom Processor to prepare the copy of the Exchange to be wire-tapped. This allows you to do any custom logic, such as deep-cloning the message payload.
cachesize		Camel 2.16 : Allows to configure the cache size for the ProducerCache which caches producers for reuse. Will by default use the default cache size which is 1000.

Setting the value to `-1` allows to turn off the cache all together.

ignoreInvalidEndpoint	<code>false</code>	Camel 2.16: Whether to ignore an endpoint URI that could not be resolved. When <code>false</code> , Camel will throw an exception when it identifies an invalid endpoint URI.
------------------------------	--------------------	---

WireTap Threadpool

The Wire Tap uses a thread pool to process the tapped messages. This thread pool will by default use the settings detailed at Threading Model. In particular, when the pool is exhausted (with all threads utilized), further wiretaps will be executed synchronously by the calling thread. To remedy this, you can configure an explicit thread pool on the Wire Tap having either a different rejection policy, a larger worker queue, or more worker threads.

WireTap Node

Camel's Wire Tap node supports two flavors when tapping an Exchange:

- With the traditional Wire Tap, Camel will copy the original Exchange and set its Exchange Pattern to `Inonly`, as we want the tapped Exchange to be sent in a fire and forget style. The tapped Exchange is then sent in a separate thread so it can run in parallel with the original. Beware that only the Exchange is copied - Wire Tap won't do a deep clone (unless you specify a custom processor via `onPrepareRef` which does that). So all copies could share objects from the original Exchange.
- Camel also provides an option of sending a new Exchange allowing you to populate it with new values.

Sending a Copy (traditional wiretap)

Using the Fluent Builders{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/WireTapTest.java}Using the Spring XML Extensions{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringWireTapTest.xml}

Sending a New Exchange

Using the Fluent Builders

Camel supports either a processor or an Expression to populate the new Exchange. Using a processor gives you full power over how the Exchange is populated as you can set properties, headers, etc. An Expression can only be used to set the IN body.

From **Camel 2.3**: the Expression or Processor is pre-populated with a copy of the original Exchange, which allows you to access the original message when you prepare a new Exchange to be sent. You can use the `copy` option (enabled by default) to indicate whether you want this. If you set `copy=false`, then it works as in **Camel 2.2** or older where the Exchange will be empty.

Below is the processor variation. This example is from **Camel 2.3**, where we disable `copy` by passing `false` to create a new, empty Exchange.{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/WireTapUsingFireAndForgetTest.java}Here is the Expression variation. In the following example we disable copying by setting `copy=false` which results in the creation of a new, empty Exchange.{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/WireTapUsingFireAndForgetTest.java}Using the Spring XML Extensions The processor variation, which uses a `processorRef` attribute to refer to a Spring bean by ID:{snippet:id=e2|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringWireTapUsingFireAndForgetTest.xml}Here is the Expression variation, where the expression is defined in the body tag:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringWireTapUsingFireAndForgetTest.xml}This variation accesses the body of the original message and creates a new Exchange based on the Expression. It will create a new Exchange and have the body contain "Bye ORIGINAL BODY MESSAGE HERE"{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringWireTapUsingFireAndForgetCopyTest.xml}

Further Example

For another example of this pattern, refer to the wire tap test case.

Using Dynamic URIs

Available as of Camel 2.16:

For example to wire tap to a dynamic URI, then it supports the same dynamic URIs as documented in Message Endpoint. For example to wire tap to a JMS queue where the header ID is part of the queue name:
`from("direct:start") .wireTap("jms:queue:backup-${header.id}") .to("bean:doSomething");`

Sending a New Exchange and Set Headers in DSL

Available as of Camel 2.8

If you send a new message using Wire Tap, then you could only set the message body using an Expression from the DSL. If you also need to set headers, you would have to use a Processor. From **Camel 2.8**: it's possible to set headers as well using the DSL.

The following example sends a new message which has

- `Bye world` as message body.
- A header with key `id` with the value `123`.
- A header with key `date` which has current date as value.

Java DSL

```
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/WireTapNewExchangeTest.java}
```

XML DSL

The XML DSL is slightly different than Java DSL in how you configure the message body and headers using `<body>` and `<setHeader>`:

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringWireTapNewExchangeTest.xml}
```

Using onPrepare to Execute Custom Logic when Preparing Messages

Available as of Camel 2.8

See details at Multicast

Using This Pattern

LOG

How can I log the processing of a Message?

Camel provides many ways to log the fact that you are processing a message. Here are just a few examples:

- You can use the Log component which logs the Message content.
- You can use the Tracer which trace logs message flow.
- You can also use a Processor or Bean and log from Java code.
- You can use the `log` DSL.

Using log DSL

In **Camel 2.2** you can use the `log` DSL which allows you to use Simple language to construct a dynamic message which gets logged.

For example you can do

```
from("direct:start").log("Processing ${id}").to("bean:foo");
```

Which will construct a String message at runtime using the Simple language. The log message will be logged at `INFO` level using the route id as the log name. By default a route is named `route-1`, `route-2` etc. But you can use the `routeId("myCoolRoute")` to set a route name of choice.

Difference between `log` in the DSL and `[Log]` component

The `log` DSL is much lighter and meant for logging human logs such as `starting to do ...` etc. It can only log a message based on the Simple language. On the other hand `Log` component is a full fledged component which involves using endpoints and etc. The `Log` component is meant for logging the Message itself and you have many URI options to control what you would like to be logged.

Using Logger instance from the Registry

As of **Camel 2.12.4/2.13.1**, if no logger name or logger instance is passed to `log` DSL, there is a Registry lookup performed to find single instance of `org.slf4j.Logger`. If such an instance is found, it is used instead of creating a new logger instance. If more instances are found, the behavior defaults to creating a new instance of logger.

Logging message body with streamed messages

If the message body is stream based, then logging the message body, may cause the message body to be *empty* afterwards. See this FAQ. For streamed messages you can use Stream caching to allow logging the message body and be able to read the message body afterwards again.

The `log` DSL have overloaded methods to set the logging level and/or name as well.

```
from("direct:start").log(LogLevel.DEBUG, "Processing ${id}").to("bean:foo");
```

and to set a logger name

```
from("direct:start").log(LogLevel.DEBUG, "com.mycompany.MyCoolRoute", "Processing ${id}").to("bean:foo");
```

Since **Camel 2.12.4/2.13.1** the logger instance may be used as well:

```
<from("direct:start").log(LoggingLevel.DEBUG, org.slf4j.LoggerFactory.getLogger("com.mycompany.mylogger"), "Processing ${id}").to("bean:foo");
```

For example you can use this to log the file name being processed if you consume files.

```
<from("file://target/files").log(LoggingLevel.DEBUG, "Processing file ${file:name}").to("bean:foo");
```

Using log DSL from Spring

In Spring DSL it is also easy to use log DSL as shown below:

```
<route id="foo">
  <from uri="direct:foo"/>
  <log message="got ${body}" />
  <to uri="mock:foo"/>
</route>
```

The log tag has attributes to set the message, loggingLevel and logName. For example:

```
<route id="baz">
  <from uri="direct:baz"/>
  <log message="Me Got ${body}" loggingLevel="FATAL" logName="com.mycompany.MyCoolRoute"/>
  <to uri="mock:baz"/>
</route>
```

Since **Camel 2.12.4/2.13.1** it is possible to reference logger instance. For example:

```
<bean id="myLogger" class="org.slf4j.LoggerFactory" factory-method="getLogger" xmlns="http://www.springframework.org/schema/beans">
  <constructor-arg value="com.mycompany.mylogger" />
</bean>

<route id="moo" xmlns="http://camel.apache.org/schema/spring">
  <from uri="direct:moo"/>
  <log message="Me Got ${body}" loggingLevel="INFO" loggerRef="myLogger"/>
  <to uri="mock:baz"/>
</route>
```

Configuring log name globally

Available as of Camel 2.17

By default the log name is the route id. If you want to use a different log name, you would need to configure the logName option. However if you have many logs and you want all of them to use the same log name, then you would need to set that logName option on all of them.

With Camel 2.17 onwards you can configure a global log name that is used instead of the route id, eg

```
CamelContext context = ...
context.getProperties().put(Exchange.LOG_EIP_NAME, "com.foo.myapp");
```

And in XML

```
<camelContext ...>
  <properties>
    <property key="CamelLogEipName" value="com.foo.myapp"/>
  </properties>
```

Using slf4j Marker

Available as of Camel 2.9

You can specify a marker name in the DSL

```
<route id="baz">
  <from uri="direct:baz"/>
  <log message="Me Got ${body}" loggingLevel="FATAL" logName="com.mycompany.MyCoolRoute" marker="myMarker"/>
  <to uri="mock:baz"/>
</route>
```

Using log DSL in OSGi

Improvement as of Camel 2.12.4/2.13.1

When using log DSL inside OSGi (e.g., in Karaf), the underlying logging mechanisms are provided by PAX logging. It searches for a bundle which invokes `org.slf4j.LoggerFactory.getLogger()` method and associates the bundle with the logger instance. Passing only logger name to log DSL results in associating `camel-core` bundle with the logger instance created.

In some scenarios it is required that the bundle associated with logger should be the bundle which contains route definition. This is possible using provided logger instance both for Java DSL and Spring DSL (see the examples above).

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 11

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 JMS Component and uses Spring's JMS support for declarative transactions, using Spring's `JmsTemplate` for sending and a `MessageListenerContainer` for consuming. All the options from the JMS component also applies for this component.

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`.

Transacted and caching

See section *Transactions and Cache Levels* below on JMS page if you are using transactions with JMS as it can impact performance.

URI format

`activemq:[queue:]topic:destinationName`

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

`activemq:FOO.BAR`

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

`activemq:queue:FOO.BAR`

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

`activemq:topic:Stocks.Prices`

Options

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

Configuring the Connection Factory

This 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

```
xmlbeans 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://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd" <camelContext
xmlns="http://camel.apache.org/schema/spring"> </camelContext> <bean id="activemq"
class="org.apache.activemq.camel.component.ActiveMQComponent"> <property name="brokerURL"
value="tcp://somehost:61616"/> </bean> </beans>
```

Using Connection Pooling

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

You can grab ActiveMQ's `org.apache.activemq.pool.PooledConnectionFactory` with Maven:

```
xml<dependency> <groupId>org.apache.activemq</groupId> <artifactId>activemq-pool</artifactId> <version>5.6.0</version>
</dependency>
```

And then setup the activemq Camel component as follows:

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

```

class="org.apache.activemq.pool.PooledConnectionFactory" init-method="start" destroy-method="stop"> <property
name="maxConnections" value="8"/> <property name="connectionFactory" ref="jmsConnectionFactory"/> </bean> <bean
id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration"> <property name="connectionFactory"
ref="pooledConnectionFactory"/> <property name="concurrentConsumers" value="10"/> </bean> <bean id="activemq"
class="org.apache.activemq.camel.component.ActiveMQComponent"> <property name="configuration" ref="jmsConfig"/> <!-- If
transacted=true then enable CACHE_CONSUMER (if not using XA) to run faster. See more details at:
http://camel.apache.org/jms --> <!-- <property name="transacted" value="true"/> <property name="cacheLevelName"
value="CACHE_CONSUMER"/> --> </bean>

```

Notice the init and destroy methods on the pooled connection factory. This is important to ensure the connection pool is properly started and shutdown.

Important information about when using transactions

If you are using transactions then see more details at JMS. And remember to set cacheLevelName to CACHE_CONSUMER if you are not using XA transactions. This can dramatically improve performance.

The PooledConnectionFactory will then create a connection pool with up to 8 connections in use at the same time. Each connection can be shared by many sessions. There is an option named maximumActive you can use to configure the maximum number of sessions per connection; the default value is 500. From **ActiveMQ 5.7**: the option has been renamed to better reflect its purpose, being named as maximumActiveSessionPerConnection. Notice the concurrentConsumers is set to a higher value than maxConnections is. This is okay, as each consumer is using a session, and as a session can share the same connection, we are in the safe. In this example we can have $8 * 500 = 4000$ active sessions at the same time.

Invoking MessageListener POJOs in a Camel route

The ActiveMQ component also provides 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);
```

That is, you can reuse any of the Camel Components and easily integrate them into your JMS MessageListener POJO!

Using ActiveMQ Destination Options

Available as of ActiveMQ 5.6

You can configure the Destination Options in the endpoint URI, using the destination. prefix. For example to mark a consumer as exclusive, and set its prefetch size to 50, you can do as follows:

```
<from uri="activemq:foo?destination.consumer.exclusive=true&&destination.consumer.prefetchSize=50"/>
```

Consuming Advisory Messages

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

The below route starts by reading the topic *ActiveMQ.Advisory.Connection*. To watch another topic, simply change the name according to the name provided in ActiveMQ Advisory Messages documentation. The parameter mapJmsMessage=false allows for converting the org.apache.activemq.command.ActiveMQMessage object from the JMS queue. Next, the body received is converted into a String for the purposes of this example and a carriage return is added. Finally, the string is added to a file

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

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

```
advisoryConnection-20100312.txt
advisoryProducer-20100312.txt
```

containing the following string:
`ActiveMQMessage { commandId = 0, responseRequired = false, messageId = ID:dell-charles-3258-1268399815140-1:0:0:221, originalDestination = null, originalTransactionId = null, producerId = ID:dell-charles-3258-1268399815140-1:0:0:0, destination = topic://ActiveMQ.Advisory.Connection, transactionId = null, expiration = 0, timestamp = 0, arrival = 0, brokerInTime = 1268403383468, brokerOutTime = 1268403383468, correlationId = null, replyTo = null, persistent = false, type = Advisory, priority = 0, groupID = null, groupSequence = 0, targetConsumerId = null, compressed = false, userID = null, content = null, marshaledProperties = org.apache.activemq.util.ByteSequence@17e2705, dataStructure = ConnectionInfo { commandId = 1, responseRequired = true, connectionId = ID:dell-charles-3258-1268399815140-2:50, clientId = ID:dell-charles-3258-1268399815140-14:0, userName = , password = *****, brokerPath = null, brokerMasterConnector = false, manageable = true, clientMaster = true }, redeliveryCounter = 0, size = 0, properties = { originBrokerName=master, originBrokerId=ID:dell-charles-3258-1268399815140-0:0, originBrokerURL=vm://master }, readOnlyProperties = true, readOnlyBody = true, droppable = false }`

Getting Component JAR

You will need this dependency

- activemq-camel

ActiveMQ is an extension of the JMS component released with the ActiveMQ project.

```
<dependency> <groupId>org.apache.activemq</groupId> <artifactId>activemq-camel</artifactId> <version>5.6.0</version>
</dependency>
```

Endpoint See Also

Unable to render {include} The included page could not be found.

AMQP

The **amqp:** component supports the AMQP 1.0 protocol using the JMS Client API of the Qpid project. In case you want to use **AMQP 0.9** (in particular RabbitMQ) you might also be interested in the Camel RabbitMQ component. Please keep in mind that prior to the **Camel 2.17.0** AMQP component supported **AMQP 0.9** and above, however since **Camel 2.17.0** it supports only **AMQP 1.0**.

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

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-amqp</artifactId>
  <version>${camel.version}</version> <!-- use the same version as your Camel core version -->
</dependency>
```

URI format

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

AMQP Options

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

Usage

As AMQP component inherits from the JMS component. The usage of the former is almost identical to the latter:

```
Listing 1. Using AMQP component

// Consuming from AMQP queue
from("amqp:queue:incoming")
  .to(...);

// Sending message to the AMQP topic
from(...)
  .to("amqp:topic:notify");
```

Configuring AMQP component

Starting from the **Camel 2.16.1** you can also use the `AMQPComponent#amqp10Component(String connectionURI)` factory method to return the **AMQP 1.0** component with the pre-configured topic prefix:

```
Listing 1. Creating AMQP 1.0 component

AMQPComponent amqp = AMQPComponent.amqp10Component("amqp://guest:guest@localhost:5672");
```

Keep in mind that starting from the **Camel 2.17** the `AMQPComponent#amqp10Component(String connectionURI)` factory method has been deprecated on the behalf of the `AMQPComponent#amqpComponent(String connectionURI)`:

```
Listing 1. Creating AMQP 1.0 component

AMQPComponent amqp = AMQPComponent.amqpComponent("amqp://localhost:5672");
AMQPComponent authorizedAmqp = AMQPComponent.amqpComponent("amqp://localhost:5672", "user", "password");
```

Starting from **Camel 2.17**, in order to automatically configure the AMQP component, you can also add an instance of `org.apache.camel.component.amqp.AMQPConnectionDetails` to the registry. For example for Spring Boot you just have to define bean:

```
Listing 1. AMQP connection details auto-configuration

@Bean
AMQPConnectionDetails amqpConnection() {
    return new AMQPConnectionDetails("amqp://localhost:5672");
}
@Bean
AMQPConnectionDetails securedAmqpConnection() {
    return new AMQPConnectionDetails("amqp://localhost:5672", "username", "password");
}
```

You can also rely on the Camel properties to read the AMQP connection details. The factory method `AMQPConnectionDetails.discoverAMQP()` attempts to read Camel properties in a Kubernetes-like convention, just as demonstrated on the snippet below:

```
Listing 1. AMQP connection details auto-configuration

export AMQP_SERVICE_HOST = "mybroker.com"
export AMQP_SERVICE_PORT = "6666"
export AMQP_SERVICE_USERNAME = "username"
export AMQP_SERVICE_PASSWORD = "password"
A
...
@Bean
AMQPConnectionDetails amqpConnection() {
    return AMQPConnectionDetails.discoverAMQP();
}
```

Configuring Connection Factory

Like with any other JMS-based component, usually it's important to configure JMS connection factory. For example, you'd like to set your broker URL or set proper connection credentials. Additionally, you would always want to set some kind of pooling (or caching) on the connection factory. An example of how to do both of these tasks is shown below.

```
<bean id="jmsConnectionFactory" class="org.apache.qpid.jms.JMsConnectionFactory">
    <property name="remoteURI" value="ampq://localhost:5672" />
    <property name="username" value="admin" />
    <property name="password" value="admin" />
</bean>

<bean id="jmsCachingConnectionFactory" class="org.springframework.jms.connection.CachingConnectionFactory">
    <property name="targetConnectionFactory" ref="jmsConnectionFactory" />
</bean>

<bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration" >
    <property name="connectionFactory" ref="jmsCachingConnectionFactory" />
    <property name="cacheLevelName" value="CACHE_CONSUMER" />
</bean>

<bean id="amqp" class="org.apache.camel.component.amqp.AMQPComponent">
    <property name="configuration" ref="jmsConfig" />
</bean>

<camelContext xmlns="http://camel.apache.org/schema/blueprint" id="simple">
    <route>
        <from uri="timer:simple?period=5000" />
        <setBody>
            <simple>Hello World</simple>
        </setBody>
        <to uri="amqp:test" />
    </route>
</camelContext>
```

Using amqp inside Karaf

To use the amqp component inside Karaf use the predefined feature called `camel-amqp` to install the necessary bundles.

Example:

```
karaf@root()> repo-add camel
karaf@root()> feature:install camel-amqp
```

and the environment would be set.

Use the `camel-blueprint` or `camel-spring` features to define routes in those contexts.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

SQS COMPONENT

Available as of Camel 2.6

The sqs component supports sending and receiving messages to Amazon's SQS service.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon SQS. More information are available at Amazon SQS.

URI Format

```
aws-sqs://queueName[?options]
aws-sqs://queueNameOrArn[?options] (from Camel 2.18)
```

The queue will be created if they don't already exists. You can append query options to the URI in the following format: ?options=value&option2=value&...

URI Options

Name	Default Value	Context	Description
accessKey	null	Shared	Amazon AWS Access Key.
amazonSQSClient	null	Shared	Reference to a com.amazonaws.services.sqs.AmazonSQS in the Registry.
amazonSQSEndpoint	null	Shared	The region with which the aws-sqs client wants to work with. Only works if Camel creates the aws-sqs client, i.e., if you explicitly set amazonSQSClient, then this setting will have no effect. You would to set it on the client you create directly
attributeNames	null	Consumer	A list of attribute names to receive when consuming. Camel 2.17: Multiple names can be separated by comma. Camel 2.16 or older: The type is a Collection so its much harder to configure and use.
concurrentConsumers	1	Consumer	(as of 2.15.0) Allows you to use multiple threads to poll the SQS queue to increase throughput. You must also set the maxMessagesPerPoll option for this to work properly.
defaultVisibilityTimeout	null	Shared	The visibility timeout (in seconds) to set in the com.amazonaws.services.sqs.model.CreateQueueRequest.
delaySeconds	null	Producer	Camel 2.9.3: Delay sending messages for a number of seconds.
deleteAfterRead	true	Consumer	Delete message from SQS after it has been read (and processed in route). If this option is false, then the same objects will be retrieved over again on the polls. Therefore you need to use the Idempotent Consumer EIP in the route to filter out duplicates. You can filter using the s3Constants#BUCKET_NAME and S3Constants#KEY headers, or only the s3Constants#KEY header.
deleteIfFiltered	true	Consumer	Camel 2.12.2, 2.13.0: Whether or not to send the deleteMessage to the SQS queue if an exchange fails to get through a filter. If false and exchange does not make it through a Camel filter upstream in the route, then don't send deleteMessage.
extendMessageVisibility	false	Consumer	Camel 2.10: If enabled a scheduled background task will keep extending the message visibility on SQS. This is needed if it takes long time to process the message. If set to true visibilityTimeout must be set. See details at Amazon docs.
maximumMessageSize	null	Shared	Camel 2.8: The maximumMessageSize (in bytes) an SQS message contain for this queue, to set in the com.amazonaws.services.sqs.model.SetQueueAttributesRequest.
maxMessagesPerPoll	null	Consumer	The maximum number of messages which can be received in one poll to set in the com.amazonaws.services.sqs.model.ReceiveMessageRequest.
messageAttributeNames	null	Consumer	A list of message attribute names to receive when consuming. Camel 2.17: Multiple names can be separated by comma. Camel 2.16 or older: The type is a Collection so its much harder to configure and use.
messageRetentionPeriod	null	Shared	Camel 2.8: The messageRetentionPeriod (in seconds) a message be retained by SQS for this queue, to set in the com.amazonaws.services.sqs.model.SetQueueAttributesRequest.
proxyHost	null	Shared	Camel 2.16: Specify a proxy host to be used inside the client definition.

proxyPort	null	Shared	Camel 2.16: Specify a proxy port to be used inside the client definition.
queueOwnerAWSAccountId	null	Shared	Camel 2.12: Specify the queue owner aws account id when you need to connect the queue with different account owner.
policy	null	shared	Camel 2.8: The policy for this queue to set in the com.amazonaws.services.sqs.model.SetQueueAttributesRequest.
receiveMessageWaitTimeSeconds	0	Shared	Camel 2.11: If you do not specify waitTimeSeconds in the request queue attribute ReceiveMessageWaitTimeSeconds is used to determine how long to wait.
redrivePolicy	null	Shared	Camel 2.15: Specify the policy that send message to deadLetter queue. See detail at Amazon docs.
region	null	Shared	Camel 2.12.3: Specify the queue region which could be used with queueOwnerAWSAccountId to build the service URL. Note: Region will still default to us-east-1 if queueOwnerAWSAccountId is not specified
secretKey	null	Shared	Amazon AWS Secret Key.
waitTimeSeconds	0	Producer	Camel 2.11: Duration in seconds (0 to 20) that the ReceiveMessage action call will wait until a message is in the queue to include in the response.
visibilityTimeout	null	Shared	The duration (in seconds) that the received messages are hidden from subsequent retrieve requests after being retrieved by a ReceiveMessage request. This only make sense if its different from defaultVisibilityTimeout.

Required SQS component options

You have to provide the amazonSQSClient in the Registry or your accesskey and secretKey to access the Amazon's SQS.

Batch Consumer

This component implements the Batch Consumer.

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

Usage

Message headers set by the SQS producer

Header	Type	Description
CamelAwsSqsMD5ofBody	String	The MD5 checksum of the Amazon SQS message.
CamelAwsSqsMessageId	String	The Amazon SQS message ID.
CamelAwsSqsDelaySeconds	Integer	Since Camel 2.11 , the delay seconds that the Amazon SQS message can be seen by others.

Message headers set by the SQS consumer

Header	Type	Description
CamelAwsSqsMD5ofBody	String	The MD5 checksum of the Amazon SQS message.
CamelAwsSqsMessageId	String	The Amazon SQS message ID.
CamelAwsSqsReceiptHandle	String	The Amazon SQS message receipt handle.
CamelAwsSqsAttributes	Map<String, String>	The Amazon SQS message attributes.

Advanced AmazonSQS configuration

If your Camel Application is running behind a firewall or if you need to have more control over the AmazonSQS instance configuration, you can create your own instance:

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);
AmazonSQS client = new AmazonSQSClient(awsCredentials, clientConfiguration);
registry.bind("client", client);
```

and refer to it in your Camel aws-sqs component configuration:

```
from("aws-sqs://MyQueue?amazonSQSClient=#client&delay=5000&maxMessagesPerPoll=5")
.to("mock:result");
```

Dependencies

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

```
Listing 1. pom.xml

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

where \${camel-version} must be replaced by the actual version of Camel (2.6 or higher).

JMS-style Selectors

SQS does not allow selectors, but you can effectively achieve this by using the Camel Filter EIP and setting an appropriate visibilityTimeout. When SQS dispatches a message, it will wait up to the visibility timeout before it will try to dispatch the message to a different consumer unless a DeleteMessage is received. By default, Camel will always send the DeleteMessage at the end of the route, unless the route ended in failure. To achieve appropriate filtering and not send the DeleteMessage even on successful completion of the route, use a Filter:

```
from("aws-sqs://MyQueue?amazonSQSClient=#client&defaultVisibilityTimeout=5000&deleteIfFiltered=false")
.filter("${header.login} == true")
.to("mock:result");
```

In the above code, if an exchange doesn't have an appropriate header, it will not make it through the filter AND also not be deleted from the SQS queue. After 5000 miliseconds, the message will become visible to other consumers.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AWS Component

ATOM COMPONENT

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

Camel will poll the feed every 60 seconds by default.

Note: The component currently only supports polling (consuming) feeds.

Maven users will need to add the following dependency to their pom.xml for this component:
`xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-atom</artifactId> <version>x.x.x</version> <!-- use the same version as your Camel core version --> </dependency>`

URI format

atom://atomUri[?options]

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

Options

confluenceTableSmall

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 <code>updateDateFilter</code> that only return new entries from the feed. So the client

consuming from the feed never receives the same entry more than once. The filter will return the entries ordered by the newest last.

lastUpdate	null	Is only used by the filter, as the starting timestamp for selection never entries (uses the entry.updated timestamp). Syntax format is: yyyy-MM-ddTHH:MM:ss. Example: 2007-12-24T17:45:59.
throttleEntries	true	Camel 2.5: Sets whether all entries identified in a single feed poll should be delivered immediately. If true, only one entry is processed per consumer.delay. Only applicable when splitEntries is set to true.
feedHeader	true	Sets whether to add the Abdera Feed object as a header.
sortEntries	false	If splitEntries is true, this sets whether to sort those entries by updated date.
consumer.delay	500	Delay in millis between each poll.
consumer.initialDelay	1000	Millis before polling starts.
consumer.userFixedDelay	false	If true, use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
username		Camel 2.16: For basic authentication when polling from a HTTP feed
password		Camel 2.16: For basic authentication when polling from a HTTP feed

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

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>.
confluenceTableSmall

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

Camel can set the Feed object on the In header (see feedHeader option to disable this):

Message Headers

Camel atom uses these headers.
confluenceTableSmall

Header	Description
CamelAtomFeed	When consuming the org.apache.abdera.model.Feed object is set to this header.

Samples

In this sample we poll James Strachan's blog.

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

In this sample we want to filter only good blogs we like to a SEDA queue. The sample also shows how to setup Camel standalone, not running in any Container or using Spring. {snippet:id=e1|lang=java|url=camel/trunk/components/camel-atom/src/test/java/org/apache/camel/component/atom/AtomGoodBlogsTest.java}Endpoint See Also

- RSS

BEAN COMPONENT

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

URI format

bean:beanID[?options]

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

Options

confluenceTableSmall

Name	Type	Default	Description
method	String	null	The method name from the bean that will be invoked. If not provided, Camel will try to determine the method itself. In case of ambiguity an exception will be thrown. See Bean Binding for more details. From Camel 2.8 onwards you

can specify type qualifiers to pin-point the exact method to use for overloaded methods. From **Camel 2.9** onwards you can specify parameter values directly in the method syntax. See more details at Bean Binding.

cache	boolean	false	If enabled, Camel will cache the result of the first Registry look-up. Cache can be enabled if the bean in the Registry is defined as a singleton scope.
multiParameterArray	boolean	false	How to treat the parameters which are passed from the message body; if it is true, the In message body should be an array of parameters.
bean.xxx	null		Camel 2.17: To configure additional options on the create bean instance from the class name. For example to configure a foo option on the bean, use bean.foo=123.

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

Using

The object instance that is used to consume messages must be explicitly registered with the Registry. For example, if you are using Spring you must define the bean in the Spring configuration, `spring.xml`; or if you don't use Spring, by registering the bean in JNDI.
 {snippet:id=register|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/pojo/PojoRouteTest.java}Once an endpoint has been registered, you can build Camel routes that use it to process exchanges.{snippet:id=route|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/pojo/PojoRouteTest.java}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:{snippet:id=invoke|lang=java|url=camel/trunk/camel-core/src/test/resources/org/apache/camel/spring/bind/beanAsEndpoint.xml}What happens is that when the exchange is routed to the `mybean` Camel will use the Bean Binding to invoke the bean.
 The source for the bean is just a plain POJO:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-spring/src/test/java/org/apache/camel/spring/bind/ExampleBean.java}Camel will use Bean Binding to invoke the `sayHello` method, by converting the Exchange's In body to the `String` type and storing the output of the method on the Exchange Out body.

Bean as endpoint

Camel also supports invoking Bean as an Endpoint. In the route below:

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/bind/beanAsEndpoint.xml}What happens is that when the exchange is routed to the mybean Camel will use the Bean Binding to invoke the bean.  

The source for the bean is just a plain POJO:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-spring/src/test/java/org/apache/camel/spring/bind/ExampleBean.java}Camel will use Bean Binding to invoke the sayHello method, by converting the Exchange's In body to the String type and storing the output of the method on the Exchange Out body.
```

Java DSL bean syntax

Java DSL comes with syntactic sugar for the Bean component. Instead of specifying the bean explicitly as the endpoint (i.e. `to("bean:beanName")`) you can use the following syntax:
`java://` Send message to the bean endpoint // and invoke method resolved using Bean Binding.
`from("direct:start").beanRef("beanName");` // Send message to the bean endpoint // and invoke given method.
`from("direct:start").beanRef("beanName", "methodName");`
 Instead of passing name of the reference to the bean (so that Camel will lookup for it in the registry), you can specify the bean itself:
`java://` Send message to the given bean instance. `from("direct:start").bean(new ExampleBean());` // Explicit selection of bean method to be invoked. `from("direct:start").bean(new ExampleBean(), "methodName");` // Camel will create the instance of bean and cache it for you. `from("direct:start").bean(ExampleBean.class);`

Bean Binding

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

Endpoint See Also

- Class component
- Bean Binding
- Bean Integration

Unable to render {include} The included page could not be found.

BROWSE COMPONENT

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

URI format

```
browse:someName[?options]
```

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

Sample

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

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

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

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

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

CACHE COMPONENT

This component is deprecated. As of Camel 2.18.0 You should use Ehcache.

Available as of Camel 2.1

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

This component supports producer and event based consumer endpoints.

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

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

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

URI format

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

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

Options

Name	Default Value	Description
<code>maxElementsInMemory</code>	1000	The number of elements that may be stored in the cache.
<code>memoryStoreEvictionPolicy</code>	<code>MemoryStoreEvictionPolicy.LFU</code>	The number of elements that may be stored in the cache. Options include: <ul style="list-style-type: none"> • <code>MemoryStoreEvictionPolicy.LFU</code> - Least freq

- MemoryStoreEvictionPolicy.LRU - Least recently used
- MemoryStoreEvictionPolicy.FIFO - first in first out, oldest element by creation time

overflowToDisk	true	Specifies whether cache may overflow to disk.
eternal	false	Sets whether elements are eternal. If eternal, timeToLiveSeconds is ignored and the element never expires.
timeToLiveSeconds	300	The maximum time between creation time and when the element expires. Is used only if the element is not eternal.
timeToIdleSeconds	300	The maximum amount of time between accesses before the element expires.
diskPersistent	false	Whether the disk store persists between restarts of the Machine.
diskExpiryThreadIntervalSeconds	120	The number of seconds between runs of the disk expiry thread.
cacheManagerFactory	null	Type: abstract org.apache.camel.component.cache.CacheManagerFactory
eventListenerRegistry	null	Camel 2.8: Sets a list of EHCache net.sf.ehcache.event.CacheEventListener for caches- no need to define it per cache in EHCache anymore.
cacheLoaderRegistry	null	Type: org.apache.camel.component.cache.CacheEventLister
key	null	Camel 2.8: Sets a list of org.apache.camel.component.cache.CacheLoaders that extends EHCache net.sf.ehcache.loader.CacheLoader for all new caches- no need to define it per cache in xml config anymore.
operation	null	Type: org.apache.camel.component.cache.CacheOperation
objectCache	false	Camel 2.10: To configure using a cache key by default provided in the message header, then the key from the operation takes precedence.
		Camel 2.10: To configure using an cache operation an operation in the message header, then the operation header takes precedence.
		Camel 2.15: Whether to turn on allowing to store non-objects in the cache. If this option is enabled then objectCache cannot be enabled as well.

Cache Component options

Name	Default Value	Description
configuration		To use a custom org.apache.camel.component.cache.CacheConfiguration configuration.
cacheManagerFactory		To use a custom org.apache.camel.component.cache.CacheManagerFactory.
configurationFile		Camel 2.13/2.12.3: To configure the location of the ehcache.xml file to use, such as classpath:com/foo/mycache.xml to load from classpath. If no configuration is given, then the default settings from EHCache is used.

Sending/Receiving Messages to/from the cache

Message Headers up to Camel 2.7

Header	Description
CACHE_OPERATION	The operation to be performed on the cache. Valid options are <ul style="list-style-type: none"> • GET

- CHECK
- ADD
- UPDATE
- DELETE
- DELETETALL
- GETANDCHECK requires Camel 2.3 onwards.

CamelKey The cache key used to store the Message in the cache. The cache key is optional if the CamelCacheOperation is DELETETALL.

Message Headers Camel 2.8+

Header changes in Camel 2.8

The header names and supported values have changed to be prefixed with 'CamelCache' and use mixed case. This makes them easier to identify and keep separate from other headers. The CacheConstants variable names remain unchanged, just their values have been changed. Also, these headers are now removed from the exchange after the cache operation is performed.

Header	Description
The operation to be performed on the cache. The valid options are	
CamelCacheoperation	<ul style="list-style-type: none"> • CamelCacheGet • CamelCacheCheck • CamelCacheAdd • CamelCacheUpdate • CamelCacheDelete • CamelCacheDeleteAll

CamelCachekey The cache key used to store the Message in the cache. The cache key is optional if the CamelCacheOperation is CamelCacheDeleteAll.

The CamelCacheAdd and CamelCacheupdate operations support additional headers:

Header	Type	Description
CamelCacheTimeToLive	Integer	Camel 2.11: Time to live in seconds.
CamelCacheTimeToIdle	Integer	Camel 2.11: Time to idle in seconds.
CamelCacheEternal	boolean	Camel 2.11: Whether the content is eternal.

Cache Producer

Sending data to the cache involves the ability to direct payloads in exchanges to be stored in a pre-existing or created-on-demand cache. The mechanics of doing this involve

- setting the Message Exchange Headers shown above.
- ensuring that the Message Exchange Body contains the message directed to the cache

Cache Consumer

Receiving data from the cache involves the ability of the CacheConsumer to listen on a pre-existing or created-on-demand Cache using an event Listener and receive automatic notifications when any cache activity take place (i.e CamelCacheGet/CamelCacheUpdate/CamelCacheDelete/CamelCacheDeleteAll). Upon such an activity taking place

- an exchange containing Message Exchange Headers and a Message Exchange Body containing the just added/updated payload is placed and sent.
- in case of a CamelCacheDeleteAll operation, the Message Exchange Header CamelCacheKey and the Message Exchange Body are not populated.

Cache Processors

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

- body
- token
- xpath level

Cache Usage Samples

Example 1: Configuring the cache

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

Example 2: Adding keys to the cache

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start")
```

```

.setHeader(CacheConstants.CACHE_OPERATION, constant(CacheConstants.CACHE_OPERATION_ADD))
.setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
.to("cache://TestCache1")
};
}

```

Example 2: Updating existing keys in a cache

```

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

```

Example 3: Deleting existing keys in a cache

```

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

```

Example 4: Deleting all existing keys in a cache

```

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

```

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

```

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

```

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

```

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

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

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

```

Example 7: Getting an entry from the Cache

```

from("direct:start")
    // Prepare headers
    .setHeader(CacheConstants.CACHE_OPERATION, constant(CacheConstants.CACHE_OPERATION_GET))
    .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
    .to("cache://TestCache1")
    // Check if entry was found
    .choice().when(header(CacheConstants.CACHE_ELEMENT_WAS_FOUND).isNull())
        .to("xf:bean:someHeavyweightOperation")
        .setHeader(CacheConstants.CACHE_OPERATION, constant(CacheConstants.CACHE_OPERATION_ADD))
        .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
        .to("cache://TestCache1")
    .end()
    .to("direct:nextPhase");

```

Example 8: Checking for an entry in the Cache

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

```

from("direct:start")
    // Prepare headers
    .setHeader(CacheConstants.CACHE_OPERATION, constant(CacheConstants.CACHE_OPERATION_CHECK))
    .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
    .to("cache://TestCache1")
    // Check if entry was not found
    .choice().when(header(CacheConstants.CACHE_ELEMENT_WAS_FOUND).isNull())
        .to("xf:bean:someHeavyweightOperation")
        .setHeader(CacheConstants.CACHE_OPERATION, constant(CacheConstants.CACHE_OPERATION_ADD))
        .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
        .to("cache://TestCache1")
    .end();

```

Management of EHCache

EHCACHE has its own statistics and management from JMX.

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

```

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

```

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

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

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

Cache replication Camel 2.8+

The Camel Cache component is able to distribute a cache across server nodes using several different replication mechanisms including: RMI, JGroups, JMS and Cache Server.

There are two different ways to make it work:

1. You can configure ehcache.xml manually
OR
2. You can configure these three options:

- cacheManagerFactory
- eventListenerRegistry
- cacheLoaderRegistry

Configuring Camel Cache replication using the first option is a bit of hard work as you have to configure all caches separately. So in a situation when the all names of caches are not known, using ehcache.xml is not a good idea.

The second option is much better when you want to use many different caches as you do not need to define options per cache. This is because replication options are set per CacheManager and per CacheEndpoint. Also it is the only way when cache names are not known at the development phase.

It might be useful to read the EHCache manual to get a better understanding of the Camel Cache replication mechanism.

Example: JMS cache replication

JMS replication is the most powerful and secured replication method. Used together with Camel Cache replication makes it also rather simple.

An example is available on a separate page.

CLASS COMPONENT

Available as of Camel 2.4

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

URI format

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

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

Options

Name	Type	Default	Description
method	String	null	The method name that bean will be invoked. If not provided, Camel will try to pick the method itself. In case of ambiguity an exception is thrown. See Bean Binding for more details.
methodParameterizing	Boolean	false	How to treat the parameters which are passed from the message body. If it is true, the In message body should be an array of parameters.
bean.xxx	null	Camel 2.17: To configure additional options on the create bean instance from the class name. For example to configure a foo option on the bean, use bean.foo=123.	

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

Using

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

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

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

```
from("direct:start").to("class:org.apache.camel.component.bean.MyFooBean?method=hello").to("mock:result");
```

SETTING PROPERTIES ON THE CREATED INSTANCE

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

```
// Camel 2.17 onwards
from("direct:start")
.to("class:org.apache.camel.component.bean.MyPrefixBean?bean.prefix=Bye")
.to("mock:result");
Â
// Camel 2.16 and olderÂ
from("direct:start")
.to("class:org.apache.camel.component.bean.MyPrefixBean?prefix=Bye")
.to("mock:result");
```

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

```
// Camel 2.17 onwards
from("direct:start")
.to("class:org.apache.camel.component.bean.MyPrefixBean?bean.cool=#foo")
.to("mock:result");
Â
// Camel 2.16 and older
from("direct:start")
.to("class:org.apache.camel.component.bean.MyPrefixBean?cool=#foo")
.to("mock:result");
```

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

See more

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

See Also

- Configuring Camel
- Component
- Endpoint

- Getting Started
- Bean
- Bean Binding
- Bean Integration

COMETD COMPONENT

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

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

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

URI format

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

The **channelName** represents a topic that can be subscribed to by the Camel endpoints.

Examples

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

where cometds: represents an SSL configured endpoint.

Options

Name	Default Value	Description
resourceBase		The root directory for the web resources or classpath. Use the protocol file: or classpath: depending if you want that the component loads the resource from file system or classpath. Classpath is required for OSGi deployment where the resources are packaged in the jar. Notice this option has been renamed to baseResource from Camel 2.7 onwards.
baseResource		Camel 2.7: The root directory for the web resources or classpath. Use the protocol file: or classpath: depending if you want that the component loads the resource from file system or classpath.
timeout	240000	The server side poll timeout in milliseconds. This is how long the server will hold a reconnect request before responding.
idleTimeout	60000	The client side poll timeout in milliseconds. A client will be removed if a connection is not received in this time.
maxInterval	30000	The max client side poll timeout in milliseconds. A client will be removed if a connection is not received in this time.
multiFrameInterval	1500	The client side poll timeout, if multiple connections are detected from the same browser.
jsonCommented	true	If true, the server will accept JSON wrapped in a comment and will generate JSON wrapped in a comment. This is a defence against Ajax Hijacking.
LogLevel	1	0:none, 1:info, 2:debug
crossoriginFilterOn	false	Camel 2.10: If true, the server will support for cross-domain filtering.
allowOrigins	*	Camel 2.10: The origins domain that support to cross if the crossorigin attribute is true.
allowCrossDomainScripting	false	Camel 2.10.5/2.11.1: (Producer only) Whether to allow cross domain scripts after publishing a message to its channel. Disconnected local sessions are needed as they are not swept by default by Camel 2.10.5/2.11.1. If false, the session can't publish a message. Camel 2.10.5/2.11.1 default value is true. Please note overriding the default value is false.
disconnectLocalSession		Camel 2.10.5/2.11.1: (Producer only) Whether to disconnect local sessions after publishing a message to its channel. Disconnecting local sessions is needed as they are not swept by default by Camel 2.10.5/2.11.1. If false, the session can't publish a message. Camel 2.10.5/2.11.1 default value is false.

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

Here is some examples on How to pass the parameters

For file (for webapp resources located in the Web Application directory --> cometd://localhost:8080?

resourceBase=file:/webapp

For classpath (when by example the web resources are packaged inside the webapp folder --> cometd://localhost:8080?

resourceBase=classpath:webapp

Authentication

Available as of Camel 2.8

You can configure custom **SecurityPolicy** and **Extension's** to the **cometdComponent** which allows you to use authentication as documented here

Setting up SSL for Cometd Component

Using the JSSE Configuration Utility

As of Camel 2.9, the Cometd component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and

component levels. The following examples demonstrate how to use the utility with the Cometd component. You need to configure SSL on the CometdComponent.

Programmatic configuration of the component

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("./users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

TrustManagersParameters tmp = new TrustManagersParameters();
tmp.setKeyStore(ksp);

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);
scp.setTrustManagers(tmp);

CometdComponent cometdComponent = getContext().getComponent("cometd", CometdComponent.class);
cometdComponent.setSslContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
<camel:sslContextParameters
    id="sslContextParameters">
    <camel:keyManagers>
        <camel:keyPassword>
            keyPassword="keyPassword"
        </camel:keyPassword>
        <camel:keyStore>
            resource="/users/home/server/keystore.jks"
            password="keystorePassword"/>
        </camel:keyStore>
    </camel:keyManagers>
    <camel:trustManagers>
        <camel:keyStore>
            resource="/users/home/server/keystore.jks"
            password="keystorePassword"/>
        </camel:keyStore>
    </camel:trustManagers>
</camel:sslContextParameters>...
<bean id="cometd" class="org.apache.camel.component.cometd.CometdComponent">
    <property name="sslContextParameters" ref="sslContextParameters"/>
</bean>
<to uri="cometd://127.0.0.1:443/service/test?baseResource=file:/target/test-classes/webapp&timeout=240000&interval=0&maxInterval=30000&mul...>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

CONTEXT COMPONENT

Available as of Camel 2.7

Deprecated do NOT use

The **context** component allows you to create new Camel Components from a CamelContext with a number of routes which is then treated as a black box, allowing you to refer to the local endpoints within the component from other CamelContexts.

It is similar to the Routebox component in idea, though the Context component tries to be really simple for end users; just a simple convention over configuration approach to refer to local endpoints inside the CamelContext Component.

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

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

URI format

```
context:camelContextId:localEndpointName[?options]
```

Or you can omit the "context:" prefix.

```
camelContextId:localEndpointName[?options]
```

- **camelContextId** is the ID you used to register the CamelContext into the Registry.
- **localEndpointName** can be a valid Camel URI evaluated within the black box CamelContext. Or it can be a logical name which is mapped to any local endpoints. For example if you locally have endpoints like **direct:invoices** and **seda:purchaseOrders** inside a CamelContext of id **supplyChain**, then you can just use the URLs **supplyChain:invoices** or **supplyChain:purchaseOrders** to omit the physical endpoint kind and use pure logical URLs.

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

Example

In this example we'll create a black box context, then we'll use it from another CamelContext.

Defining the context component

First you need to create a CamelContext, add some routes in it, start it and then register the CamelContext into the Registry (JNDI, Spring, Guice or OSGi etc).

This can be done in the usual Camel way from this test case (see the `createRegistry()` method); this example shows Java and JNDI being used...

```
// lets create our black box as a camel context and a set of routes
DefaultCamelContext blackBox = new DefaultCamelContext(registry);
blackBox.setName("blackBox");
blackBox.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        // receive purchase orders, lets process it in some way then send an invoice
        // to our invoice endpoint
        from("direct:purchaseOrder").
            setHeader("received").constant("true").
            to("direct:invoice");
    }
});
blackBox.start();
registry.bind("accounts", blackBox);
```

Notice in the above route we are using pure local endpoints (**direct** and **seda**). Also note we expose this CamelContext using the **accounts** ID. We can do the same thing in Spring via

```
<camelContext id="accounts" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:purchaseOrder"/>
        ...
        <to uri="direct:invoice"/>
    </route>
</camelContext>
```

Using the context component

Then in another CamelContext we can then refer to this "accounts black box" by just sending to **accounts:purchaseOrder** and consuming from **accounts:invoice**.

If you prefer to be more verbose and explicit you could use **context:accounts:purchaseOrder** or even **context:accounts:direct://purchaseOrder** if you prefer. But using logical endpoint URIs is preferred as it hides the implementation detail and provides a simple logical naming scheme.

For example if we wish to then expose this accounts black box on some middleware (outside of the black box) we can do things like...

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <!-- consume from an ActiveMQ into the black box -->
        <from uri="activemq:Accounts.PurchaseOrders"/>
        <to uri="accounts:purchaseOrders"/>
    </route>
    <route>
        <!-- lets send invoices from the black box to a different ActiveMQ Queue -->
        <from uri="accounts:invoice"/>
        <to uri="activemq:UK.Accounts.Invoices"/>
    </route>
</camelContext>
```

Naming endpoints

A context component instance can have many public input and output endpoints that can be accessed from outside its CamelContext. When there are many it is recommended that you use logical names for them to hide the middleware as shown above.

However when there is only one input, output or error/dead letter endpoint in a component we recommend using the common posix shell names **in**, **out** and **err**

CRYPTO COMPONENT FOR DIGITAL SIGNATURES

Available as of Camel 2.3

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

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

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

Introduction

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

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

- Bruce Schneier's Applied Cryptography
- Beginning Cryptography with Java by David Hook
- The ever insightful Wikipedia Digital_signatures

URI format

As mentioned Camel provides a pair of crypto endpoints to create and verify signatures
`crypto:sign:name[?options] crypto:verify:name[?options]`

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

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

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

Options

confluenceTableSmall

Name	Type	Default	Description
algorithm	String	SHA1withDSA	The name of the JCE Signature algorithm that will be used.
alias	String	null	An alias name that will be used to select a key from the keystore.
bufferSize	Integer	2048	the size of the buffer used in the signature process.
certificate	Certificate	null	A Certificate used to verify the signature of the exchange's payload. Either this or a Public Key is required.
keystore	KeyStore	null	A reference to a JCE Keystore that stores keys and certificates used to sign and verify.
keyStoreParameters	KeyStoreParameters	null	A reference to a Camel KeyStoreParameters Object which wraps a Java KeyStore Object
Camel 2.14.1			
provider	String	null	The name of the JCE Security Provider that should be used.
privateKey	PrivateKey	null	The private key used to sign the exchange's payload.
publickey	PublicKey	null	The public key used to verify the signature of the

			exchange's payload.
secureRandom	secureRandom	null	A reference to a <code>SecureRandom</code> object that will be used to initialize the Signature service.
password	char[]	null	The password to access the private key from the keystore
clearHeaders	String	true	Remove camel crypto headers from Message after a verify operation (value can be "true"/"false").

Using

1) Raw keys

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

```
{snippet:id=basic|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}The same can be achieved with the Spring XML Extensions using references to keys{snippet:id=basic|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}
```

2) KeyStores and Aliases.

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

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

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

```
{snippet:id=keystore|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}Again in Spring a ref is used to lookup an actual keystore instance.{snippet:id=keystore|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}
```

3) Changing JCE Provider and Algorithm

Changing the Signature algorithm or the Security provider is a simple matter of specifying their names. You will need to also use Keys that are compatible with the algorithm you choose.{snippet:id=algorithm|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}{snippet:id=provider|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}or{snippet:id=algorithm|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}{snippet:id=provider|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}

4) Changing the Signature Message Header

It may be desirable to change the message header used to store the signature. A different header name can be specified in the route definition as follows{snippet:id=signature-header|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}or{snippet:id=signature-header|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}

5) Changing the buffersize

In case you need to update the size of the buffer...{snippet:id=buffersize|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}or{snippet:id=buffersize|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}

6) Supplying Keys dynamically.

When using a Recipient list or similar EIP the recipient of an exchange can vary dynamically. Using the same key across all recipients may be neither feasible nor desirable. It would be useful to be able to specify signature keys dynamically on a per-exchange basis. The exchange could then be dynamically enriched with the key of its target recipient prior to signing. To facilitate this the signature mechanisms allow for keys to be supplied dynamically via the message headers below

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

{snippet:id=headerkey|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java} or {snippet:id=headerkey|lang=xml|url=camel/trunk/components/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml} Even better would be to dynamically supply a keystore alias. Again the alias can be supplied in a message header

- Exchange.KEYSTORE_ALIAS, "CamelSignatureKeystoreAlias"

{snippet:id=alias|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java} or {snippet:id=alias|lang=xml|url=camel/trunk/components/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml} The header would be set as follows

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

Endpoint See Also

- Crypto Crypto is also available as a Data Format

CXF COMPONENT

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

When using CXF in streaming modes (see DataFormat option), then also read about Stream caching.

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:

```
xml<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>
```

If you want to learn about CXF dependencies you can checkout the WHICH-JARS text file.

URI format

javacxf:bean:cxfEndpoint[?options]

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

javacxf:/someAddress[?options]

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

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

javacxf:bean:cxfEndpoint?wsdlURL=wsdl/hello_world.wsdl&dataFormat=PAYOUT

Options

Name	Required	Description
wsdlURL	No	<p>The location of the WSDL. It is obtained from endpoint address by default.</p> <p><i>Example:</i> file:///local/wsdl/hello.wsdl or wsdl1/he11o.wsdl</p>
serviceClass	Yes	<p>The name of the SEI (Service Endpoint Interface) class. This class can have, but c annotations.</p> <p>This option is only required by POJO mode. If the wsdlURL option is provided, ser for PAYLOAD and MESSAGE mode. When wsdlURL option is used without service and portName (endpointName for Spring configuration) options MUST be provided notation to reference a serviceClass object instance from the registry. E.g. servi serviceClass for a CXF producer (that is, the to endpoint) should be a Java inter Since 2.8, it is possible to omit both wsdlURL and serviceClass options for PAYLO When they are omitted, arbitrary XML elements can be put in CxfPayload's body ir facilitate CXF Dispatch Mode.</p>
serviceName	No	<p>Please be advised that the referenced object cannot be a Proxy (Spring AOP Pr object.getClass().getName() method for non Spring AOP Proxy).</p> <p><i>Example:</i> org.apache.camel.Hello</p> <p>The service name this service is implementing, it maps to the wsdl : serviceName</p>

		Required for camel-cxf consumer since camel-2.2.0 or if more than one serviceName
		<i>Example:</i> {http://org.apache.camel}ServiceName
		The port name this service is implementing, it maps to the wsdl:port@name.
endpointName	No	Required for camel-cxf consumer since camel-2.2.0 or if more than one portName serviceName.
		<i>Example:</i> {http://org.apache.camel}PortName
		The data type messages supported by the CXF endpoint.
dataFormat	No	<i>Default:</i> POJO <i>Example:</i> POJO, PAYLOAD, MESSAGE
relayHeaders	No	Please see the Description of relayHeaders option section for this option. Shoot headers along the route. Currently only available when dataFormat=POJO
		<i>Default:</i> true <i>Example:</i> true, false
wrapped	No	Which kind of operation that CXF endpoint producer will invoke
wrappedStyle	No	New in 2.5.0 The WSDL style that describes how parameters are represented in t is false, CXF will chose the document-literal unwrapped style, If the value is true, C document-literal wrapped style
		<i>Default:</i> null <i>Example:</i> true, false
setDefaultBus	No	Deprecated Will set the default bus when CXF endpoint create a bus by itself. This defaultBus from Camel 2.16 onwards.
		<i>Default:</i> false <i>Example:</i> true, false
defaultBus	No	Camel 2.16: Will set the default bus when CXF endpoint create a bus by itself
		<i>Default:</i> false <i>Example:</i> true, false
bus	No	A default bus created by CXF Bus Factory. Use # notation to reference a bus object. referenced object must be an instance of org.apache.cxf.Bus.
		<i>Example:</i> bus=#busName
cxfBinding	No	Use # notation to reference a CXF binding object from the registry. The referenced instance of org.apache.camel.component.cxf.CxfBinding (use an instance of org.apache.camel.component.cxf.DefaultCxfBinding).
		<i>Example:</i> cxfbinding=#bindingName
headerFilterStrategy	No	Use # notation to reference a header filter strategy object from the registry. The ref instance of org.apache.camel.spi.HeaderFilterStrategy (use an instance of org.apache.camel.component.cxf.CxfHeaderFilterStrategy).
		<i>Example:</i> headerFilterStrategy=#strategyName
loggingFeatureEnabled	No	New in 2.3. This option enables CXF Logging Feature which writes inbound and out to log.
		<i>Default:</i> false <i>Example:</i> loggingFeatureEnabled=true
defaultOperationName	No	New in 2.4, this option will set the default operationName that will be used by the C the remote service.
		<i>Default:</i> null <i>Example:</i> defaultOperationName=greetMe
defaultOperationNamespace	No	New in 2.4. This option will set the default operationNamespace that will be used b invokes the remote service.
		<i>Default:</i> null <i>Example:</i> defaultoperationNamespace=http://apache.org/hello_world_soap
synchronous	No	New in 2.5. This option will let cxf endpoint decide to use sync or async API to do t

		default value is false which means camel-cxf endpoint will try to use async API by
		<i>Default:</i> false <i>Example:</i> synchronous=true
publishedEndpointUrl	No	New in 2.5. This option can override the endpointUrl that published from the WSDL with service address url plus ?wsdl.
		<i>Default:</i> null <i>Example:</i> publishedEndpointUrl=http://example.com/service
properties.XXX	No	Camel 2.8: Allows to set custom properties to CXF in the endpoint uri. For example properties.mtom-enabled=true to enable MTOM. properties.org.apache.cxf.interceptor.OnewayProcessorInterceptor.us just make sure the CXF doesn't switch the thread when start the invocation.
allowStreaming	No	New in Camel 2.8.2 . This option controls whether the CXF component, when runn (see below), will DOM parse the incoming messages into DOM Elements or keep them as javax.xml.transform.Source object that would allow streaming in some cases.
skipFaultLogging	No	New in Camel 2.11 . This option controls whether the PhaseInterceptorChain skips catches.
cxEndpointConfigurer	No	New in Camel 2.11 . This option could apply the implementation of org.apache.camel.component.cxf.CxfEndpointConfigurer which supports to endpoint in programmatic way. Since Camel 2.15.0 , user can configure the CXF's implementing configure{Server Client} method of CxfEndpointConfigurer.
username	No	New in Camel 2.12.3 This option is used to set the basic authentication information for client.
password	No	New in Camel 2.12.3 This option is used to set the basic authentication information for client.
continuationTimeout	No	New in Camel 2.14.0 This option is used to set the CXF continuation timeout which CxfConsumer by default when the CXF server is using Jetty or Servlet transport. (If CxfConsumer just set the continuation timeout to be 0, which means the continuation never timeout.) <i>Default:</i> 30000 <i>Example:</i> continuation=80000
cookieHandler	No	New in Camel 2.19.0 : Configure a cookie handler to maintain a HTTP session.

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. Both Protocol and Logical JAX-WS handlers are supported.
PAYLOAD	PAYOUTLOAD is the message payload (the contents of the soap:body) after message configuration in the CXF endpoint is applied. Only Protocol JAX-WS handler is supported. Logical JAX-WS handler is not supported.
MESSAGE	MESSAGE is the raw message that is received from the transport layer. It is not suppose to touch or change Stream, some of the CXF interceptors will be removed if you are using this kind of DataFormat so you can't see any soap headers after the camel-cxf consumer and JAX-WS handler is not supported.
CXF_MESSAGE	New in Camel 2.8.2 , CXF_MESSAGE allows for invoking the full capabilities of CXF interceptors by converting the message from the transport layer into a raw SOAP message

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

How to enable CXF's LoggingOutInterceptor in MESSAGE mode

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

```
{snippet:id:enableLoggingOutInterceptor|lang=xml|url=camel/trunk/components/camel-cxf/src/test/resources/org/apache/camel/component/cxf/LoggingInterceptorInMessageModeTest-context.xml}
```

Description of relayHeaders option

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

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

The *out-of-band* headers are headers that are serialized over the wire, but are not explicitly part of the WSDL binding contract. Headers relaying/filtering is bi-directional.

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

Available only in POJO mode

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

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

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

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

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

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

Changes since Release 2.0

- POJO and PAYLOAD modes are supported. In POJO mode, only out-of-band message headers are available for filtering as the in-band headers have been processed and removed from header list by CXF. The in-band headers are incorporated into the `MessageContentList` in POJO mode. The `camel-cxf` component does make any attempt to remove the in-band headers from the `MessageContentList`. If filtering of in-band headers is required, please use PAYLOAD mode or plug in a (pretty straightforward) CXF interceptor/JAXWS Handler to the CXF endpoint.
- The Message Header Relay mechanism has been merged into `CxfHeaderFilterStrategy`. The `relayHeaders` option, its semantics, and default value remain the same, but it is a property of `CxfHeaderFilterStrategy`.
Here is an example of configuring it.

```
{snippet:id=dropAllMessageHeadersStrategy|lang=xml|url=camel/trunk/components/camel-cxf/src/test/resources/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTest-context.xml}Then, your endpoint can reference the CxfHeaderFilterStrategy.  

{snippet:id=noRelayRoute|lang=xml|url=camel/trunk/components/camel-cxf/src/test/resources/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTest-context.xml}
```
- The `MessageHeadersRelay` interface has changed slightly and has been renamed to `MessageHeaderFilter`. It is a property of `CxfHeaderFilterStrategy`. Here is an example of configuring user defined Message Header Filters:

```
{snippet:id=customMessageFilterStrategy|lang=xml|url=camel/trunk/components/camel-cxf/src/test/resources/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTest-context.xml}
```
- Other than `relayHeaders`, there are new properties that can be configured in `CxfHeaderFilterStrategy`.

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

Configure the CXF endpoints with Spring

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

```
xml<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:cxf="http://camel.apache.org/schema/cxf" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd"> <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://camel.apache.org/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 <code>wsdl:port@name</code> . In the format of <code>ns:PORT_NAME</code> where <code>ns</code> is a namespace prefix valid at this scope.
serviceName	The service name this service is implementing, it maps to the <code>wsdl:service@name</code> . In the format of <code>ns:SERVICE_NAME</code> where <code>ns</code> 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 <code>bindingId</code> for the service model to use.
address	The service publish address.
bus	The bus name that will be used in the JAX-WS endpoint.
serviceClass	The class name of the SEI (Service Endpoint Interface) class which could have JSR181 annotation or not.

It also supports many child elements:

Name	Value
<code>cxf:inInterceptors</code>	The incoming interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
<code>cxf:inFaultInterceptors</code>	The incoming fault interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
<code>cxf:outInterceptors</code>	The outgoing interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
<code>cxf:outFaultInterceptors</code>	The outgoing fault interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
<code>cxf:properties</code>	A properties map which should be supplied to the JAX-WS endpoint. See below.
<code>cxf:handlers</code>	A JAX-WS handler list which should be supplied to the JAX-WS endpoint. See below.
<code>cxf:dataBinding</code>	You can specify the which databinding will be use in the endpoint. This can be supplied using the Spring <code><bean class="MyDatabinding"/></code> syntax.
<code>cxf:binding</code>	You can specify the BindingFactory for this endpoint to use. This can be supplied using the Spring <code><bean class="MyBindingFactory"/></code> syntax.
<code>cxf:features</code>	The features that hold the interceptors for this endpoint. A list of <code>{<bean>}s</code> or <code>{<ref>}s</code>
<code>cxf:schemaLocations</code>	The schema locations for endpoint to use. A list of <code>{<schemaLocation>}s</code>
<code>cxf:serviceFactory</code>	The service factory for this endpoint to use. This can be supplied using the Spring <code><bean class="MyServiceFactory"/></code> syntax

You can find more advanced examples that show how to provide interceptors, properties and handlers on the CXF JAX-WS Configuration page.

NOTE

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

```
xml<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>
```

Configuring the CXF Endpoints with Apache Aries Blueprint.

Since camel 2.8 there is support for utilizing aries blueprint dependency injection for your CXF endpoints. The schema utilized is very similar to the spring schema so the transition is fairly transparent.

Example

```
xml<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0" xmlns:camel-
  cxf="http://camel.apache.org/schema/blueprint/cxf" xmlns:cxfcore="http://cxf.apache.org/blueprint/core"
  xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">
<camel-cxf:cxfEndpoint id="routerEndpoint" address="http://localhost:9001/router"
```

```
serviceClass="org.apache.servicemix.examples.cxf.HelloWorld"> <camel-cxf:properties> <entry key="dataFormat"
```

```
value="MESSAGE"/> </camel-cxf:properties> </camel-cxf:cxfEndpoint> <camel-cxf:cxfEndpoint id="serviceEndpoint"
```

```
address="http://localhost:9000/SoapContext/SOAPPort" serviceClass="org.apache.servicemix.examples.cxf.HelloWorld">
```

```
</camel-cxf:cxfEndpoint> <camelContext xmlns="http://camel.apache.org/schema/blueprint"> <route> <from
```

```
uri="routerEndpoint"/> <to uri="log:request"/> </route> </camelContext> </blueprint>
```

Currently the endpoint element is the first supported CXF namespacehandler.

You can also use the bean references just as in spring

```
xml<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0" xmlns:jaxws="http://cxf.apache.org/blueprint/jaxws"
  xmlns:cxf="http://cxf.apache.org/blueprint/core" xmlns:camel="http://camel.apache.org/schema/blueprint"
  xmlns:camelcxf="http://camel.apache.org/schema/blueprint/cxf" xsi:schemaLocation=" http://www.osgi.org/xmlns/blueprint/v1.0.0
  https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd http://cxf.apache.org/blueprint/jaxws
  http://cxf.apache.org/schemas/blueprint/jaxws.xsd http://cxf.apache.org/blueprint/core
  http://cxf.apache.org/schemas/blueprint/core.xsd "> <camelcxf:cxfEndpoint id="reportIncident" address="/camel-example-cxf-
  blueprint/webservices/incident" wsdlURL="META-INF/wsdl/report_incident.wsdl"
```

```
serviceClass="org.apache.camel.example.reportincident.ReportIncidentEndpoint"> </camelcxf:cxfEndpoint> <bean
```

```
id="reportIncidentRoutes" class="org.apache.camel.example.reportincident.ReportIncidentRoutes" /> <camelContext
```

```
xmlns="http://camel.apache.org/schema/blueprint"> <routeBuilder ref="reportIncidentRoutes"/> </camelContext> </blueprint>
```

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

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

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

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

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

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

```
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-
cxf/src/test/java/org/apache/camel/component/cxf/WriteXmlDeclarationInterceptor.java}You can add a customer interceptor like
this and configure it into your camel-cxf endpoint{snippet:id=example|lang=xml|url=camel/trunk/components/camel-
cxf/src/test/resources/org/apache/camel/component/cxf/GreeterEndpointsRouterContext.xml}Or adding a message header for it
like this if you are using Camel 2.4.
```

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

How to override the CXF producer address from message header

The camel-cxf producer supports to override the services address by setting the message with the key of `"CamelDestinationOverrideUrl"`.

```
// set up the service address from the message header to override the setting of CXF endpoint
exchange.getIn().setHeader(Exchange.DESTINATION_OVERRIDE_URL, constant(getServiceAddress()));
```

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

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

```
{snippet:id=personProcessor|lang=java|url=camel/trunk/components/camel-
cxf/src/test/java/org/apache/camel/wsdl_first/PersonProcessor.java}
```

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 to a list, and initialize the message with this parameter list. The response message's body is a `messageContentsList`, you can get the result from that list.

If you don't specify the operation name in the message header, CxfProducer will try to use the `defaultOperationName` from `CxfEndpoint`, if there is no `defaultOperationName` set on `CxfEndpoint`, it will pickup the first `operationName` from the Operation list.

If you want to get the object array from the message body, you can get the body using `message.getBody(Object[].class)`, as follows:{snippet:id=sending|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/CxfProducerRouterTest.java}

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

PAYLOAD means that you process the payload message from the SOAP envelope. You can use the `Header.HEADER_LIST` as the key to set or get the SOAP headers and use the `List<Element>` to set or get SOAP body elements.

`Message.getBody()` will return an `org.apache.camel.component.cxf.cxfPayLoad` object, which has getters for SOAP

message headers and Body elements. This change enables decoupling the native CXF message from the Camel message.

```
{snippet:id=payload|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/CxfConsumerPayloadTest.java}
```

How to get and set SOAP headers in POJO mode

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

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

```
{snippet:id=processSoapHeaderRoute|lang=xml|url=camel/trunk/components/camel-cxf/src/test/resources/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTest-context.xml}SOAP headers are propagated to and from Camel Message headers. The Camel message header name is "org.apache.cxf.headers.Header.list" which is a constant defined in CXF (org.apache.cxf.headers.Header.HEADER_LIST). The header value is a List of CXF SoapHeader objects (org.apache.cxf.binding.soap.SoapHeader). The following snippet is the InsertResponseOutHeaderProcessor (that insert a new SOAP header in the response message). The way to access SOAP headers in both InsertResponseOutHeaderProcessor and InsertRequestOutHeaderProcessor are actually the same. The only difference between the two processors is setting the direction of the inserted SOAP header.
```

```
{snippet:id=InsertResponseOutHeaderProcessor|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTest.java}
```

How to get and set SOAP headers in PAYLOAD mode

We've already shown how to access SOAP message (`CxfPayLoad` object) in PAYLOAD mode (See "How to deal with the message for a camel-cxf endpoint in PAYLOAD data format").

Once you obtain a `CxfPayLoad` object, you can invoke the `CxfPayLoad.getHeaders()` method that returns a List of DOM Elements (SOAP headers).{snippet:id=payload|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/CxfPayLoadSoapHeaderTest.java}Since Camel 2.16.0, you can also use the same way as described in sub-chapter "How to get and set SOAP headers in POJO mode" to set or get the SOAP headers. So, you can use now the header "org.apache.cxf.headers.Header.list" to get and set a list of SOAP headers. This does also mean that if you have a route that forwards from one Camel-cxf endpoint to another (SOAP Client -> Camel -> CXF service), now also the SOAP headers sent by the SOAP client are forwarded to the CXF service. If you do not want that these headers are forwarded you have to remove them in the Camel header "org.apache.cxf.headers.Header.list".

SOAP headers are not available in MESSAGE mode

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

How to throw a SOAP Fault from Camel

If you are using a camel-cxf endpoint to consume the SOAP request, you may need to throw the SOAP Fault from the camel context.

Basically, you can use the `throwFault` DSL to do that; it works for POJO, PAYLOAD and MESSAGE data format.

You can define the soap fault like this{snippet:id=FaultDefine|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/CxfCustomizedExceptionTest.java}Then throw it as you like{snippet:id=ThrowFault|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/CxfCustomizedExceptionTest.java}If your CXF endpoint is working in the MESSAGE data format, you could set the the SOAP Fault message in the message body and set the response code in the message header.{snippet:id=MessageStreamFault|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/CxfMessageStreamExceptionTest.java}Same for using POJO data format. You can set the `SOAPFault` on the out body and also indicate it's a fault by calling `Message.setFault(true)`:

```
{snippet:id=onException|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/CxfMessageStreamExceptionTest.java}
```

How to propagate a 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 a camel-cxf endpoint producer to invoke the outside web service, you can set the request context and get response context with the following code:

```
java 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 first element of the array is the return value Object[] output = out.getBody(Object[].class); LOG.info("Received output text: " + output[0]); // Get the response context form outMessage Map<String, Object> responseContext = CastUtils.cast((Map)out.getHeader(Client.RESPONSE_CONTEXT)); assertNotNull(responseContext); assertEquals("Get the wrong wsdl operation name", "
```

{http://apache.org/hello_world_soap_http/greetMe", responseContext.get("javax.xml.ws.wsdl.operation").toString());

Attachment Support

POJO Mode: Both SOAP with Attachment and MTOM are supported (see example in Payload Mode for enabling MTOM). However, SOAP with Attachment is not tested. Since attachments are marshalled and unmarshalled into POJOs, users typically do not need to deal with the attachment themselves. Attachments are propagated to Camel message's attachments if the MTOM is not enabled, since 2.12.3. So, it is possible to retrieve attachments by Camel Message API DataHandler Message.getAttachment(String id)

Payload Mode: MTOM is supported since 2.1. Attachments can be retrieved by Camel Message APIs mentioned above. SOAP with Attachment (SwA) is supported and attachments can be retrieved since 2.5. SwA is the default (same as setting the CXF endpoint property "mtom-enabled" to false).

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

```
{snippet:id=enableMtom|lang=xml|url=camel/trunk/components/camel-cxf/src/test/resources/org/apache/camel/component/cxf/mtom/CxfMtomRouterPayloadModeTest-context.xml}You can produce a Camel message with attachment to send to a CXF endpoint in Payload mode.
```

```
{snippet:id=producer|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/mtom/CxfMtomProducerPayloadModeTest.java}You can also consume a Camel message received from a CXF endpoint in Payload mode.
```

```
{snippet:id=consumer|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/mtom/CxfMtomConsumerPayloadModeTest.java}Message Mode: Attachments are not supported as it does not process the message at all.
```

CXF_MESSAGE Mode: MTOM is supported, and Attachments can be retrieved by Camel Message APIs mentioned above. Note that when receiving a multipart (i.e. MTOM) message the default SOAPMessage to String converter will provide the complete multipart payload on the body. If you require just the SOAP XML as a String, you can set the message body with message.getSOAPPart(), and Camel convert can do the rest of work for you.

Streaming Support in PAYLOAD mode

In 2.8.2, the camel-cxf component now supports streaming of incoming messages when using PAYLOAD mode. Previously, the incoming messages would have been completely DOM parsed. For large messages, this is time consuming and uses a significant amount of memory. Starting in 2.8.2, the incoming messages can remain as a javax.xml.transform.Source while being routed and, if nothing modifies the payload, can then be directly streamed out to the target destination. For common "simple proxy" use cases (example: from("cxf...").to("cxf...")), this can provide very significant performance increases as well as significantly lowered memory requirements.

However, there are cases where streaming may not be appropriate or desired. Due to the streaming nature, invalid incoming XML may not be caught until later in the processing chain. Also, certain actions may require the message to be DOM parsed anyway (like WS-Security or message tracing and such) in which case the advantages of the streaming is limited. At this point, there are two ways to control the streaming:

- Endpoint property: you can add "allowStreaming=false" as an endpoint property to turn the streaming on/off.
- Component property: the CxfComponent object also has an allowStreaming property that can set the default for endpoints created from that component.

Global system property: you can add a system property of "org.apache.camel.component.cxf.streaming" to "false" to turn it off. That sets the global default, but setting the endpoint property above will override this value for that endpoint.

Using the generic CXF Dispatch mode

From 2.8.0, the camel-cxf component supports the generic CXF dispatch mode that can transport messages of arbitrary structures (i.e., not bound to a specific XML schema). To use this mode, you simply omit specifying the wsdlURL and serviceClass attributes of the CXF endpoint.

```
xml<cxf:cxfEndpoint id="testEndpoint" address="http://localhost:9000/SapContext/SapAnyPort"> <cxf:properties> <entry key="dataFormat" value="PAYLOAD"/> </cxf:properties> </cxf:cxfEndpoint>
```

It is noted that the default CXF dispatch client does not send a specific SOAPAction header. Therefore, when the target service requires a specific SOAPAction value, it is supplied in the Camel header using the key SOAPAction (case-insensitive).

Endpoint See Also

CXF BEAN COMPONENT

The **cxfbean:** component allows other Camel endpoints to send exchange and invoke Web service bean objects. **Currently, it only supports JAX-RS and JAX-WS (new to Camel 2.1) annotated service beans.**

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

Maven users need to add the following dependency to their pom.xml to use the CXF Bean Component:

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

URI format

cxfbean:serviceBeanRef

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

Options

confluenceTableSmall

Name	Description	Example
bus	CXF bus reference specified by the # notation. The referenced object must be an instance of org.apache.cxf.Bus.	bus=#busName
cxfBeanBinding	CXF bean binding specified by the # notation. The referenced object must be an instance of org.apache.camel.component.cxf.cxfbean.CxfBeanBinding.	cxfBinding=#bindingName
headerFilterStrategy	Header filter strategy specified by the # notation. The referenced object must be an instance of org.apache.camel.spi.HeaderFilterStrategy.	headerFilterStrategy=#strategy
populateFromClass	Since 2.3, the wsdlLocation annotated in the POJO is ignored (by default) unless this option is set to false. Prior to 2.3, the wsdlLocation annotated in the POJO is always honored and it is not possible to ignore.	true, false
providers	Since 2.5, setting the providers for the CXFRS endpoint.	providers=#providerRef1,#providerRef2
setDefaultBus	Will set the default bus when CXF endpoint create a bus by itself.	true, false

Headers

confluenceTableSmall

Name	Description	Type	Required?	Default Value	In/Out	Examples
CamelHttpCharacterEncoding (before 2.0-m2: CamelCxfBeanCharacterEncoding)	Character encoding	String	No	None	In	ISO-8859-1
CamelContentType (before 2.0-m2: CamelCxfBeanContentType)	Content type	String	No	/*	In	text/xml
CamelHttpBaseUri (2.0-m3 and before: CamelCxfBeanRequestBasePath)	The value of this header will be set in the CXF message as the Message.BASE_PATH property. It is needed by CXF JAX-RS processing. Basically, it is the scheme, host and	String	Yes	The Endpoint URI of the source endpoint in the Camel exchange	In	http://localhost:9000/SapContext/SapAnyPort

port portion of the request URI.						
	Request URI's path	String	Yes	None	In	consumer/123
CamelHttpMethod (before 2.0-m2: CamelCxfbeanVerb)	RESTful request verb	String	Yes	None	In	GET, PUT, POST, DELETE
CamelHttpResponseCode	HTTP response code	Integer	No	None	Out	200

Currently, the CXF Bean component has (only) been tested with the Jetty component. It understands headers from Jetty component without requiring conversion.

A Working Sample

This sample shows how to create a route that starts an embedded Jetty HTTP server. The route sends requests to a CXF Bean and invokes a JAX-RS annotated service.

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

```
{snippet:id=routeDefinition|lang=xml|url=camel/trunk/components/camel-cxf/src/test/resources/org/apache/camel/component/cxf/cxfbean/CxfBeanTest-context.xml}
```

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

```
{snippet:id=beanDefinition|lang=xml|url=camel/trunk/components/camel-cxf/src/test/resources/org/apache/camel/component/cxf/cxfbean/CxfBeanTest-context.xml}
```

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

CXFRS COMPONENT

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

The `cxfrs:` component provides integration with Apache CXF for connecting to JAX-RS 1.1 and 2.0 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

`javacfrs://address?options`

Where `address` represents the CXF endpoint's address

`javacfrs:bean:rsEndpoint`

Where `rsEndpoint` represents the spring bean's name which presents the CXFRS client or server

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

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

Options

Name	Description	Example	R
resourceClasses	The resource classes which you want to export as REST service. Multiple classes can be separated by comma.	<code>resourceClasses=org.apache.camel.rs.Example1, org.apache.camel.rs.Exchange2</code>	N
resourceClass	Deprecated: Use <code>resourceClasses</code> The resource class which you want to export as REST service.	<code>resourceClass =org.apache.camel.rs.Example1</code>	N
httpClientAPI	new to Camel 2.1 If it is true, the CxfRsProducer will use the HttpClientAPI to invoke the service If it is false, the	<code>httpClientAPI=true</code>	N

	CxfRsProducer will use the ProxyClientAPI to invoke the service		
synchronous	<p>This option will let you decide to use sync or async API to do the underlying work. The default value is false which means it will try to use async API by default.</p> <p>This option is available as of 2.5 for CxfRsConsumer and as of 2.19 for CxfRsProducer.</p>	synchronous=true	N
throwExceptionOnFailure	New in 2.6, this option tells the CxfRsProducer to inspect return codes and will generate an Exception if the return code is larger than 207.	throwExceptionOnFailure=true	N
maxClientCachesize	New in 2.6, you can set a IN message header CamelDestinationOverrideUrl to dynamically override the target destination Web Service or REST Service defined in your routes. The implementation caches CXF clients or ClientFactoryBean in CxfProvider and CxfRsProvider. This option allows you to configure the maximum size of the cache.	maxClientCacheSize=5	N
setDefaultBus	New in 2.9.0. deprecated use defaultBus option from Camel 2.16 onwards. Will set the default bus when CXF endpoint create a bus by itself	setDefaultBus=true	N
defaultBus	Camel 2.16: Will set the default bus when CXF endpoint create a bus by itself	defaultBus=true	N
bus	New in 2.9.0. A default bus created by CXF Bus Factory. Use # notation to reference a bus object from the registry. The referenced object must be an instance of org.apache.cxf.bus.	bus=#busName	N
bindingstyle	<p>As of 2.11. Sets how requests and responses will be mapped to/from Camel. Two values are possible:</p> <ul style="list-style-type: none"> • simpleConsumer => see the Consuming a REST Request with the Simple Binding Style below. • Default => the default style. For consumers this passes on a MessageContentsList to the route, requiring low-level processing in the route. • Custom => allows you to specify a custom binding through the binding option. 	bindingStyle=SimpleConsumer	N
binding	Allows you to specify a	binding=#myBinding	N

	custom cxfRsBinding implementation to perform low-level processing of the raw CXF request and response objects. The implementation must be bound in the Camel registry, and you must use the hash (#) notation to refer to it.		
providers	Since Camel 2.12.2 set custom JAX-RS providers list to the CxfRs endpoint.	providers=#MyProviders	N
schemaLocations	Since Camel 2.12.2 Sets the locations of the schemas which can be used to validate the incoming XML or JAXB-driven JSON.	schemaLocations=#MySchemaLocations	N
features	Since Camel 2.12.3 Set the feature list to the CxfRs endpoint.	features=#MyFeatures	N
properties	Since Camel 2.12.4 Set the properties to the CxfRs endpoint.	properties=#MyProperties	N
inInterceptors	Since Camel 2.12.4 Set the inInterceptors to the CxfRs endpoint.	inInterceptors=#MyInterceptors	N
outInterceptors	Since Camel 2.12.4 Set the outInterceptor to the CxfRs endpoint.	outInterceptors=#MyInterceptors	N
inFaultInterceptors	Since Camel 2.12.4 Set the inFaultInterceptors to the CxfRs endpoint.	inFaultInterceptors=#MyInterceptors	N
outFaultInterceptors	Since Camel 2.12.4 Set the outFaultInterceptors to the CxfRs endpoint.	outFaultInterceptors=#MyInterceptors	N
continuationTimeout	Since Camel 2.14.0 This option is used to set the CXF continuation timeout which could be used in CxfConsumer by default when the CXF server is using Jetty or Servlet transport. (Before Camel 2.14.0, CxfConsumer just set the continuation timeout to be 0, which means the continuation suspend operation never timeout.)	continuationTimeout=800000	N
ignoreDeleteMethodMessageBody	Since Camel 2.14.1 This option is used to tell CxfRsProducer to ignore the message body of the DELETE method when using HTTP API.	ignoreDeleteMethodMessageBody=true	N
modelRef	Since Camel 2.14.2 This option is used to specify the model file which is useful for	modelRef=classpath:/CustomerServiceModel.xml	N

	the resource class without annotation.	
	Since Camel 2.15 This option can point to a model file without specifying a service class for emulating document-only endpoints	
performInvocation	Since Camel 2.15 When the option is true, camel will perform the invocation of the resource class instance and put the response object into the exchange for further processing.	performInvocation= true
propagateContexts	Since Camel 2.15 When the option is true, JAXRS UriInfo, HttpHeaders, Request and SecurityContext contexts will be available to custom CXFRS processors as typed Camel exchange properties. These contexts can be used to analyze the current requests using JAX-RS API.	N
loggingFeatureEnabled	This option enables CXF Logging Feature which writes inbound and outbound REST messages to log.	N
skipFaultLogging	This option controls whether the PhaseInterceptorchain skips logging the Fault that it catches.	N
loggingSizeLimit	To limit the total size of number of bytes the logger will output when logging feature has been enabled.	N
cookieHandler	Since Camel 2.19.0 Configure a cookie handler to maintain a HTTP session	cookieHandler=#exchangeCookieHandler

You can also configure the CXF REST endpoint through the spring configuration. Since there are lots of difference between the CXF REST client and CXF REST Server, we provide different configuration for them.

Please check out the schema file and CXF JAX-RS documentation for more information.

How to configure the REST endpoint in Camel

In camel-cxf schema file, there are two elements for the REST endpoint definition. **cxfrs:rsServer** for REST consumer, **cxfrs:rsClient** for REST producer.

You can find a Camel REST service route configuration example here.

```
{snippet:id=cxfRsExample|lang=xml|url=camel/trunk/components/camel-cxf/src/test/resources/org/apache/camel/component/cxf/jaxrs/CxfRsSpringRouter.xml}
```

How to override the CXF producer address from message header

The camel-cxf producer supports to override the services address by setting the message with the key of "CamelDestinationOverrideUrl".

```
// set up the service address from the message header to override the setting of CXF endpoint
exchange.getIn().setHeader(Exchange.DESTINATION_OVERRIDE_URL, constant(getServiceAddress()));
```

Consuming a REST Request - Simple Binding Style

Available as of Camel 2.11

The default binding style is rather low-level, requiring the user to manually process the `MessageContentsList` object coming into the route. Thus, it tightly couples the route logic with the method signature and parameter indices of the JAX-RS operation. Somewhat inelegant, difficult and error-prone.

In contrast, the `SimpleConsumer` binding style performs the following mappings, in order to make the request data more accessible to you within the Camel Message:

- JAX-RS Parameters (@HeaderParam, @QueryParam, etc.) are injected as IN message headers. The header name matches the value of the annotation.
- The request entity (POJO or other type) becomes the IN message body. If a single entity cannot be identified in the JAX-RS method signature, it falls back to the original MessageContentsList.
- Binary @Multipart body parts become IN message attachments, supporting DataHandler, InputStream, DataSource and CXF's Attachment class.
- Non-binary @Multipart body parts are mapped as IN message headers. The header name matches the Body Part name.

Additionally, the following rules apply to the **Response mapping**:

- If the message body type is different to javax.ws.rs.core.Response (user-built response), a new Response is created and the message body is set as the entity (so long it's not null). The response status code is taken from the Exchange.HTTP_RESPONSE_CODE header, or defaults to 200 OK if not present.
- If the message body type is equal to javax.ws.rs.core.Response, it means that the user has built a custom response, and therefore it is respected and it becomes the final response.
- In all cases, Camel headers permitted by custom or default HeaderFilterStrategy are added to the HTTP response.

Enabling the Simple Binding Style

This binding style can be activated by setting the bindingStyle parameter in the consumer endpoint to value simpleConsumer:
java from("cxf:bean:rsServer?bindingStyle=SimpleConsumer").to("log:TEST?showAll=true");

Examples of request binding with different method signatures

Below is a list of method signatures along with the expected result from the Simple binding.

```
public Response doAction(Businessobject request);
Request payload is placed in IN message body, replacing the original MessageContentsList.
public Response doAction(Businessobject request, @HeaderParam("abcd") String abcd, @QueryParam("defg")
String defg);
Request payload placed in IN message body, replacing the original MessageContentsList. Both request params mapped as IN
message headers with names abcd and defg.
public Response doAction(@HeaderParam("abcd") String abcd, @QueryParam("defg") String defg);
Both request params mapped as IN message headers with names abcd and defg. The original MessageContentsList is
preserved, even though it only contains the 2 parameters.
public Response doAction(@Multipart(value="body1") Businessobject request, @Multipart(value="body2")
Businessobject request2);
The first parameter is transferred as a header with name body1, and the second one is mapped as header body2. The original
MessageContentsList is preserved as the IN message body.
public Response doAction(InputStream abcd);
The InputStream is unwrapped from the MessageContentsList and preserved as the IN message body.
public Response doAction(DataHandler abcd);
The DataHandler is unwrapped from the MessageContentsList and preserved as the IN message body.
```

More examples of the Simple Binding Style

Given a JAX-RS resource class with this method:

```
java @POST @Path("/customers/{type}") public Response newCustomer(Customer customer, @PathParam("type") String type,
@QueryParam("active") @DefaultValue("true") boolean active) { return null; }
```

Serviced by the following route:

```
java from("cxf:bean:rsServer?bindingStyle=SimpleConsumer").recipientList(simple("direct:${header.operationName}"));
from("direct:newCustomer").log("Request: type=${header.type}, active=${header.active}, customerData=${body}");
```

The following HTTP request with XML payload (given that the Customer DTO is JAXB-annotated):

```
xmlPOST /customers/gold?active=true Payload: <Customer> <fullName>Raul Kripalani</fullName> <country>Spain</country>
<project>Apache Camel</project> </Customer>
```

Will print the message:

```
xmlRequest: type=gold, active=true, customerData=<Customer> toString() representation>
```

For more examples on how to process requests and write responses can be found here.

Consuming a REST Request - Default Binding Style

The CXF JAXRS front end implements the JAX-RS (JSR-311) API, so we can export the resources classes as a REST service. And we leverage the CXF Invoker API to turn a REST request into a normal Java object method invocation.

Unlike the Camel Restlet component, you don't need to specify the URI template within your endpoint, CXF takes care of the REST request URI to resource class method mapping according to the JSR-311 specification. All you need to do in Camel is delegate this method request to a right processor or endpoint.

Here is an example of a CXFRS route...{snippet:id=example|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/jaxrs/CxfRsConsumerTest.java}And the corresponding resource class used to configure the endpoint...

Note about resource classes

By default, JAX-RS resource classes are **only** used to configure JAX-RS properties. Methods will **not** be executed during routing of messages to the endpoint. Instead, it is the responsibility of the route to do all processing.

Note that starting from Camel 2.15 it is also sufficient to provide an interface only as opposed to a no-op service implementation class for the default mode.

Starting from Camel 2.15, if a **performInvocation** option is enabled, the service implementation will be invoked first, the response will be set on the Camel exchange and the route execution will continue as usual. This can be useful for

integrating the existing JAX-RS implementations into Camel routes and for post-processing JAX-RS Responses in custom processors.

```
{snippet:id=example|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/jaxrs/testbean/CustomerServiceResource.java}
```

How to invoke the REST service through camel-cxfrs producer

The CXF JAXRS front end implements a proxy-based client API, with this API you can invoke the remote REST service through a proxy. The camel-cxfrs producer is based on this proxy API.

You just need to specify the operation name in the message header and prepare the parameter in the message body, the camel-cxfrs producer will generate right REST request for you.

Here is an example:{snippet:id=ProxyExample|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/jaxrs/CxfRsProducerTest.java}The CXF JAXRS front end also provides a http centric client API. You can also invoke this API from camel-cxfrs producer. You need to specify the **HTTP_PATH** and the **HTTP_METHOD** and let the producer use the http centric client API by using the URI option **httpClientAPI** or by setting the message header **CxfConstants.CAMEL_CXF_RS_USING_HTTP_API**. You can turn the response object to the type class specified with the message header **CxfConstants.CAMEL_CXF_RS_RESPONSE_CLASS**.

```
{snippet:id=HttpExample|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/jaxrs/CxfRsProducerTest.java}From Camel 2.1, we also support to specify the query parameters from cxfrs URI for the CXFRS http centric client.
```

```
{snippet:id=QueryExample|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/jaxrs/CxfRsProducerTest.java}To support the Dynamical routing, you can override the URI's query parameters by using the CxfConstants.CAMEL_CXF_RS_QUERY_MAP header to set the parameter map for it.{snippet:id=QueryMapExample|lang=java|url=camel/trunk/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/jaxrs/CxfRsProducerTest.java}
```

DATASET COMPONENT

Testing of distributed and asynchronous processing is notoriously difficult. The Mock, Test and DataSet endpoints work great with the Camel Testing Framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's large range of Components together with the powerful Bean Integration.

The DataSet component 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.

Camel will use the throughput logger when sending dataset's.

URI format

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

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

Camel ships with a support implementation of **org.apache.camel.component.dataset.DataSet**, the **org.apache.camel.component.dataset.DataSetSupport** class, that can be used as a base for implementing your own **DataSet**.

Camel also ships with some implementations that can be used for testing:

- **org.apache.camel.component.dataset.SimpleDataSet**
- **org.apache.camel.component.dataset.ListDataSet**
- **org.apache.camel.component.dataset.FileDataSet**

all of which extend **DataSetSupport**.

Options

Option	Default	Description
produceDelay	3	Allows a delay in ms to be specified, which causes producers to pause in order to simulate slow producers. Uses a minimum of 3ms.delay . Set to -1 to force no delay at all.
consumeDelay	0	Allows a delay in ms to be specified, which causes consumers to pause in order to simulate slow consumers.
preLoadSize	0	Sets how many messages should be pre-loaded (sent) before the route completes its initialization.
initialDelay	1000	Camel 2.1: Time period in milliseconds to wait before starting sending messages.
minRate	0	Wait until the dataset contains at least this number of messages.
datasetIndex	1/next	Camel 2.17: Controls the behavior of the CamelDatasetIndex header.

The supported values are:

- strict
- lenient
- off

The default behavior prior to Camel 2.17 can be restored using `datasetIndex=strict`.

Client Type	datasetIndex Value	Camel datasetIndex Header Behavior
Consumer	strict	The header will always be set.
	lenient	The header will NOT be set.
	off	The header will NOT be set.
Producer	strict	If the header is present, the value of the header will be verified.
	lenient	If the header is present, the value of the header will not be verified. If the header is not present, it will be set.
	off	If the header is present, the value of the header will not be verified. If the header is not present, it will not be set.

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

Configuring DataSet

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

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

Example

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

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

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

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

DATASETSUPPORT (ABSTRACT CLASS)

The `DataSetSupport` abstract class is a nice starting point for new `DataSets`, and provides some useful features to derived classes.

Properties on DataSetSupport

Property	Type	Default	Description
<code>defaultHeaders</code>	<code>Map<String, Object></code>	<code>null</code>	Specifies the default message body. For <code>SimpleDataSet</code> it is a constant payload; though if you want to create custom payloads per message, create your own derivation of <code>DataSetSupport</code> .
<code>outputProcessor</code>	<code>Processor</code>	<code>null</code>	
<code>size</code>	<code>long</code>	<code>10</code>	Specifies how many messages to send/consume.
<code>reportCount</code>	<code>long</code>	<code>-1</code>	Specifies the number of messages to be received before reporting progress. Useful for showing progress of a large load test. If <code>-1</code> If = 0 then size Else set to reportCount value.

SimpleDataSet

The `SimpleDataSet` extends `DataSetSupport`, and adds a default body.

Additional Properties on SimpleDataSet

Property	Type	Default	Description
<code>defaultBody</code>	<code>Object</code>	<code><helloWorld!></code>	Specifies the default message body. By default, the <code>SimpleDataSet</code> produces the same constant payload for each exchange. If you want to customize the payload for each exchange, create a <code>Camel Processor</code> and configure the <code>SimpleDataSet</code> to use it by setting the <code>outputTransformer</code> property.

ListDataSet (Camel 2.17)

The `ListDataSet` extends `DataSetSupport`, and adds a list of default bodies.

Additional Properties on ListDataSet

Property	Type	Default	Description
<code>defaultBodies</code>	<code>List<Object></code>	<code>empty</code>	Specifies the default message body. By default, the <code>ListDataSet</code> selects a constant payload from the list of <code>defaultBodies</code> using the <code>datasetIndex</code> . If you want to customize the payload, create a <code>Camel Processor</code> and configure the <code>ListDataSet</code> to use it by setting the <code>outputTransformer</code> property.
<code>size</code>	<code>long</code>	the size of	Specifies how many messages to send/consume. This value can be different from the size of the <code>defaultBodies</code> list. If the value is less than the size of the <code>defaultBodies</code> list,

the defaultbodies list some of the list elements will not be used. If the value is greater than the size of the defaultbodies list, the payload for the exchange will be selected using the modulus of the CamelDataSetIndex and the size of the defaultbodies list (i.e. CamelDataSetIndex % defaultbodies.size())

FileDataSet (Camel 2.17)

The simpleDataSet extends ListDataSet, and adds support for loading the bodies from a file.

Additional Properties on FileDataSet

Property	Type	Default	Description
sourcefile	File	null	Specifies the source file for payloads
delimiter	String	\z	Specifies the delimiter pattern used by a java.util.Scanner to split the file into multiple payloads.

See Also

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

DB4O COMPONENT

Available as of Camel 2.5

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

Sending to the endpoint

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

Consuming from the endpoint

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

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

URI format

db4o:className[?options]

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

Options

Name	Default Value	Description
consumeDelete	true	Option for db4oConsumer only. Specifies whether or not the entity is deleted after it is consumed.
consumer.delay	500	Option for consumer only. Delay in millis between each poll.
consumer.initialDelay	1000	Option for consumer only. Millis before polling starts.
consumer.userFixedDelay	false	Option for consumer only. Set to true to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

DIRECT COMPONENT

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

Asynchronous

The SEDA component provides asynchronous invocation of any consumers when a producer sends a message exchange. Connection to other camel contexts
The VM component provides connections between Camel contexts as long they run in the same JVM.

URI format

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

Where someName can be any string that uniquely identifies the endpoint.

Options

Name	Default Value	Description
allowMultipleConsumers	true	@deprecated If set to false, then when a second consumer is started on the endpoint, an <code>IllegalStateException</code> is thrown. Will be removed in Camel 2.1: Direct endpoint does not support multiple consumers.
block	false	Camel 2.11.1: If sending a message to a direct endpoint which has no active consumer, the producer will block for <code>timeout</code> milliseconds waiting for a consumer to become active.
timeout	30000	Camel 2.11.1: The timeout value, in milliseconds, to block, when enabled, for an active consumer.
failIfNoConsumers	true	Camel 2.16.0: Indicates whether the producer should fail by throwing an exception when sending to a direct endpoint with no active consumers.

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

Samples

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

```
from("activemq:queue:order.in")
  .to("bean:orderServer?method=validate")
  .to("direct:processOrder?block=true&timeout=5000");

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

And the sample using spring DSL:

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

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

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

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- SEDA
- VM

DNS

Available as of Camel 2.7

This is an additional component for Camel to run DNS queries, using DNSJava. The component is a thin layer on top of DNSJava.

The component offers the following operations:

- ip, to resolve a domain by its ip
- lookup, to lookup information about the domain
- dig, to run DNS queries

Requires SUN JVM

The DNSJava library requires running on the SUN JVM.

If you use Apache ServiceMix or Apache Karaf, you'll need to adjust the etc/jre.properties file, to add sun.net.spi.nameservice to the list of Java platform packages exported. The server will need restarting before this change takes effect.

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

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

URI format

The URI scheme for a DNS component is as follows

```
dns://operation[?options]
```

This component only supports producers.

Options

None.

Headers

Header	Type	Operations	Description
dns.domain	String	ip	The domain name. Mandatory.
dns.name	String	lookup	The name to lookup. Mandatory.
dns.type	String	lookup, dig	The type of lookup. Should match the values of org.apache.camel.dns.type. Optional.
dns.class	String	lookup, dig	The DNS class of the lookup. Should match the values of org.apache.camel.dns.class. Optional.
dns.query	String	dig	The query itself. Mandatory.
dns.server	String	dig	The server in particular for the query. If none is given, the default one specified by the OS will be used. Optional.

Examples

IP lookup

```
<route id="IPCheck">
<from uri="direct:start"/>
<to uri="dns:ip"/>
</route>
```

This looks up a domain's IP. For example, www.example.com resolves to 192.0.32.10.

The IP address to lookup must be provided in the header with key "dns.domain".

DNS lookup

```
<route id="IPCheck">
<from uri="direct:start"/>
<to uri="dns:lookup"/>
</route>
```

This returns a set of DNS records associated with a domain.

The name to lookup must be provided in the header with key "dns.name".

DNS Dig

Dig is a Unix command-line utility to run DNS queries.

```
<route id="IPCheck">
<from uri="direct:start"/>
```

```
<to uri="dns:dig"/>
</route>
```

The query must be provided in the header with key "dns.query".

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

EJB COMPONENT

Available as of Camel 2.4

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

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

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

URI format

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

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

Options

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

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

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

Bean Binding

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

Examples

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

```
Listing 1. GreaterLocal.java
public interface GreaterLocal {
    String hello(String name);
    String bye(String name);
}
```

And the implementation

```
Listing 1. GreaterImpl.java
@Stateless
public class GreaterImpl implements GreaterLocal {
    public String hello(String name) {
        return "Hello " + name;
    }
    public String bye(String name) {
        return "Bye " + name;
    }
}
```

Using Java DSL

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

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

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

    return answer;
}

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

    return new InitialContext(properties);
}
```

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

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

In a real application server

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

Using Spring XML

And this is the same example using Spring XML instead:

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

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

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

Before we are ready to use EJB in the Camel routes:

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

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

- Bean
- Bean Binding
- Bean Integration

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[?options]
```

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

Pattern example:

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

EQL example:

```
from("esper://esper-dom?eql=insert into DomStream select * from org.w3c.dom.Document")
.to("log://esper-dom?level=INFO");
from("esper://esper-dom?eql=select childNodes from DomStream")
.to("mock:results");
```

Options

Name	Default Value	Description
configured	false	Available as of camel-extra 2.11: If flag is set to 'true' the default Esper configuration file (esper-dg.xml) will be used. If flag is set to 'false' no Esper configuration file, please refer to the Esper documentation
eql		The Esper EQL expression as a String to filter events

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

EsperMessage

From **Camel 2.12** onwards the esper consumer stores new and old events in the `org.apacheextras.camel.component.esper.EsperMessage` message as the input Message on the Exchange. You can get access to the esper event beans from java code with:

```
EventBean newEvent = exchange.getIn(EsperMessage.class).getNewEvent();
EventBean oldEvent = exchange.getIn(EsperMessage.class).getOldEvent();
```

By default if you get the body of `org.apacheextras.camel.component.esper.EsperMessage` it returns the new `EventBean` as in previous versions.

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

Unable to render {include} The included page could not be found.

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 to be saved to disk.

URI format

`file:directoryName[?options]`

or

`file://directoryName[?options]`

Where `directoryName` represents the underlying file directory.

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

Only directories

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

Avoid reading files currently being written by another application

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

URI Options

Common

confluenceTableSmall

Name	Default Value	Description
<code>autoCreate</code>	<code>true</code>	Automatically create missing directories in the file's path name. For the file consumer, that means creating the starting directory. For the file producer, it means the directory the files should be written to.
<code>buffersize</code>	<code>128kb</code>	Write buffer sized in bytes.
<code>fileName</code>	<code>null</code>	<p>Use Expression such as File Language to dynamically set the filename. For consumers, it's used as a filename filter. For producers, it's used to evaluate the filename to write. If an expression is set, it takes precedence over the <code>CamelFileName</code> header. (Note: The header itself can also be an Expression). The expression options support both string and Expression types. If the expression is a String type, it is always evaluated using the File Language. If the expression is an Expression type, the specified Expression type is used - this allows you, for instance, to use OGNL expressions.</p> <p>For the consumer, you can use it to filter filenames, so you can for instance consume today's file using the File Language syntax: <code>mydata-\${date:now:yyyyMMdd}.txt</code>. From Camel 2.11 onward the producers support the <code>CamelOverruleFileName</code> header which takes precedence over any existing <code>CamelFileName</code> header; the <code>CamelOverruleFileName</code> is a header that is used only once, and makes it easier as this avoids to temporary store <code>CamelFileName</code> and have to restore it afterwards.</p>
<code>flatten</code>	<code>false</code>	Flatten is used to flatten the file name path to strip any leading paths, so it's just the file name. This allows you to consume recursively into sub-directories, but when you eg write the files to another directory they will be written in a single directory. Setting this to true on the producer enforces that any file name received in <code>CamelFileName</code> header will be stripped for any leading paths.
<code>charset</code>	<code>null</code>	Camel 2.9.3: this option is used to specify the encoding of the file. You can use this on the consumer, to specify the encodings of the files, which allow Camel to know the charset it should load the file content in case the file content is being accessed. Likewise when writing a file, you can use this option to specify which charset to write the file as well. See further below for examples and more important details.
<code>copyAndDeleteOnRenameFail</code>	<code>true</code>	Camel 2.9: whether to fallback and do a copy and delete file, in case the file could not be renamed directly. This option is not available for the FTP component.
<code>renameUsingCopy</code>	<code>false</code>	Camel 2.13.1: Perform rename operations using a copy and delete strategy. This is primarily used in environments where the regular rename operation is unreliable e.g., across different file systems or networks. This option takes precedence over the <code>copyAndDeleteOnRenameFail</code> parameter that will automatically fall back to the copy and delete strategy, but only after additional delays.

Consumer

confluenceTableSmall

Name	Default Value	Description
<code>initialDelay</code>	<code>1000</code>	Milliseconds before polling the file/directory starts.
<code>delay</code>	<code>500</code>	Milliseconds before the next poll of the file/directory.
<code>useFixedDelay</code>		Controls if fixed delay or fixed rate is used. See <code>ScheduledExecutorS</code> for details. In Camel 2.7.x or older the default value is <code>false</code> . From Camel 2.8 onward the default value is <code>true</code> .
<code>runLoggingLevel</code>	<code>TRACE</code>	Camel 2.8: The consumer logs a start/complete log line when it polls

		- allows you to configure the logging level for that.
recursive	false	If a directory, will look for files in all the sub-directories as well.
delete	false	If true, the file will be deleted after it is processed successfully.
noop	false	If true, the file is not moved or deleted in any way. This option is good for data, or for ETL type requirements. If noop=true, Camel will set idempotent as well, to avoid consuming the same files over and over again.
preMove	null	Expression (such as File Language) used to dynamically set the filename moving it before processing. For example to move in-progress files in a directory set this value to order.
move	.camel	Expression (such as File Language) used to dynamically set the filename moving it after processing. To move files into a .done subdirectory just use .done.
moveFailed	null	Expression (such as File Language) used to dynamically set a different directory when moving files in case of processing (configured via move above) failed. For example, to move files into a .error subdirectory use: .error Note: When moving the files to the "fail" location Camel will handle them and will not pick up the file again.
include	null	Is used to include files, if filename matches the regex pattern (match insensitive from Camel 2.17 onward).
exclude	null	Is used to exclude files, if filename matches the regex pattern (match insensitive from Camel 2.17 onward).
antInclude	null	Camel 2.10: Ant style filter inclusion, for example antInclude=*/.txt inclusions may be specified in comma-delimited format. See below for more details about ant path filters.
antExclude	null	Camel 2.10: Ant style filter exclusion. If both antInclude and antExclude are used, antExclude takes precedence over antInclude. Multiple excludes can be specified in comma-delimited format. See below for more details about ant path filters.
antFilterCaseSensitive	true	Camel 2.11: Ant style filter which is case sensitive or not.
idempotent	false	Option to use the Idempotent Consumer EIP pattern to let Camel skip processed files. Will by default use a memory based LRU Cache that stores entries. If noop=true then idempotent will be enabled as well to avoid the same files over and over again.
idempotentKey	Expression	Camel 2.11: To use a custom idempotent key. By default the absolute file name is used. You can use the File Language, for example to use the file size, you can do: <code>idempotentKey=\${file:name}-\${file:size}</code>
idempotentRepository	null	A pluggable repository org.apache.camel.spi.IdempotentRepository. Default use MemoryMessageIdRepository if none is specified and idempotent is true.
inProgressRepository	memory	A pluggable in-progress repository org.apache.camel.spi.IdempotentRepository. The in-progress repository is used to account the current in progress files being consumed. By default a memory based repository is used.
filter	null	Pluggable filter as a org.apache.camel.component.file.GenericFileFilter class. Will skip files if filter returns false in its accept() method. More details below.
filterDirectory	null	Camel 2.18: Filters the directory based on Simple language. For example on current date, you can use a simple date pattern such as \${date:r}
filterFile	null	Camel 2.18: Filters the file based on Simple language. For example if file size, you can use \${file:size}>5000.
shuffle	false	Camel 2.16: To shuffle the list of files (sort in random order).
sorter	null	Pluggable sorter as a java.util.Comparator<org.apache.camel.component.file.GenericFile> class.
sortBy	null	Built-in sort using the File Language. Supports nested sorts, so you can sort by file name and as a 2nd group sort by modified date. See sorting section for details.
readLock	none	Used by consumer, to only poll the files if it has exclusive read-lock on the file is not in-progress or being written. Camel will wait until the file is granted. This option provides the built-in strategies:

- none is for no read locks at all.
- **markerFile** Camel creates a marker file `fileName.camelLoc` holds a lock on it. This option is **not** available for the FTP component. changed is using file length/modification timestamp to detect whether a file is currently being copied or not. Will at least use 1 sec. to detect this option cannot consume files as fast as the others, but can be reliable as the JDK IO API cannot always determine whether a file is currently being used by another process. The option `readLockCheckInterval` can be used to set the check frequency option is **only** available for the FTP component from **Camel 2.8** onwards. from **Camel 2.10.1** onward the FTP option `fastExistsCheck` is enabled to speedup this readLock strategy, if the FTP server supports a LIST operation with a full file name (some servers may not).
- **fileLock** is for using `java.nio.channels.FileLock`. This option is available for the FTP component. This approach should be avoided for accessing a remote file system via a mount/share unless that component supports distributed file locks.
- **rename** is for using a try to rename the file as a test if we can get a read-lock.
- **idempotent** **Camel 2.16** (only file component) is for using a `idempotentRepository` as the read-lock. This allows to use a `idempotentRepository` that supports clustering if the idempotent repository implementation supports that.
- **idempotent-changed** **Camel 2.19** (only file component) is for using a `idempotentRepository` and changed as combined read-lock. This allows to use a `idempotentRepository` that supports clustering if the idempotent repository implementation supports that.
- **idempotent-rename** **Camel 2.19** (only file component) is for using a `idempotentRepository` and rename as combined read-lock. This allows to use a `idempotentRepository` that supports clustering if the idempotent repository implementation supports that.

Warning: most of the read lock strategies are not suitable for use in clustered mode. That is, you cannot have multiple consumers attempting to read the same file in the same directory. In this case, the read locks will not function correctly. idempotent read lock supports clustered reliably if you use a cluster aware idempotent repository implementation such as from Hazelcast or Infinispan.

		Optional timeout in milliseconds for the readLock, if supported. If the lock could not be granted and the timeout triggered, then Camel will skip the current poll, Camel will try the file again, and this time maybe the read-lock could be granted. Use a value of 0 or lower to indicate forever. In Camel 2.0 the default value is 0. Starting with Camel 2.1 the default value is 10000. Currently changed and rename support the timeout.
<code>readLockTimeout</code>	10000	Note: for FTP the default <code>readLockTimeout</code> value is 20000 instead of 0. The <code>readLockTimeout</code> value must be higher than <code>readLockCheckInterval</code> . The rule of thumb is to have a timeout that is at least 2 or more times higher than <code>readLockCheckInterval</code> . This is needed to ensure that ample time is given for the read lock process to try to grab the lock before the timeout was reached.
<code>readLockCheckInterval</code>	1000	Camel 2.6: Interval in milliseconds for the read-lock, if supported by the component. This interval is used for sleeping between attempts to acquire the read lock. For example when using the changed read lock, you can set a higher interval to cater for slow writes. The default of 1 sec. may be <i>too fast</i> if the producer is slow writing the file. For FTP the default <code>readLockCheckInterval</code> is 1000ms. The <code>readLockTimeout</code> value must be higher than <code>readLockCheckInterval</code> . The rule of thumb is to have a timeout that is at least 2 or more times higher than the <code>readLockCheckInterval</code> . This is needed to ensure that ample time is given for the read lock process to try to grab the lock before the timeout was reached.
<code>readLockMinLength</code>	1	Camel 2.10.1: This option applies only for <code>readLock=changed</code> . This option allows you to configure a minimum file length. By default Camel expects the file to contain some data, and thus the default value is 1. You can set this option to zero, if you want to consume zero-length files.
<code>readLockMinAge</code>	0	Camel 2.15: This option applies only to <code>readLock=change</code> . This option allows you to specify a minimum age a file must be before attempting to acquire the read lock. For example, use <code>readLockMinAge=300s</code> to require that the file is at least 300 minutes old. This can speedup the poll when the file is old enough as the read lock immediately. Notice for FTP: file timestamps reported by the file system are often reported with resolution of minutes, so <code>readLockMinAge</code> parameter is not very accurate.

		should be defined in minutes, e.g. 60000 for 1 minute. Notice that Camel 2.12 specifies this as 60s, or 1m, etc.
readLockLoggingLevel	WARN	<p>Camel 2.12: Logging level used when a read lock could not be acquired. A WARN is logged. You can change this level, for example to OFF to no logging.</p> <p>This option is only applicable for the readLock types:</p> <ul style="list-style-type: none"> • changed • fileLock • rename
readLockMarkerFile	true	<p>Camel 2.14: Whether to use marker file with the changed, rename, or read lock types. By default a marker file is used as well to guard against processes picking up the same files. This behavior can be turned off by setting to false. For example if you do not want to write marker files to the file system by the Camel application.</p>
readLockRemoveOnRollback	true	<p>Camel 2.16: This option applied only for readLock=idempotent. This is to specify whether to remove the file name entry from the idempotent when processing the file failed and a rollback happens. If this option is true the file name entry is confirmed (as if the file did a commit).</p>
readLockRemoveOnCommit	false	<p>Camel 2.16: This option applied only for readLock=idempotent. This is to specify whether to remove the file name entry from the idempotent when processing the file succeeded and a commit happens. By default not removed which ensures that any race-condition do not occur so a node may attempt to grab the file. Instead the idempotent repository uses eviction strategies that you can configure to evict the file name entry after 10 minutes - this ensures no problems with race conditions.</p>
readLockDeleteOrphanLockFiles	true	<p>Camel 2.16: Whether or not read lock with marker files should be deleted upon any orphan read lock files, which may have been left on the file system after a shutdown (such as a JVM crash). If turning this option off then any orphaned lock file will cause Camel to not attempt to pickup files. This could also be due another node is concurrently reading files from the directory.</p>
directoryMustExist	false	<p>Camel 2.5: Similar to startingDirectoryMustExist but this applies to polling recursive sub directories.</p>
doneFileName	null	<p>Camel 2.6: If provided, Camel will only consume files if a <i>done</i> file exists. This option configures what file name to use. Either you can specify a fixed file name or use dynamic placeholders. The <i>done</i> file is always expected to be in the folder as the original file. See using done file and writing done file section for examples.</p>
exclusiveReadLockStrategy	null	<p>Pluggable read-lock as a <code>org.apache.camel.component.file.GenericFileExclusiveReadStrategy</code> implementation.</p>
maxMessagesPerPoll	0	<p>An integer to define a maximum messages to gather per poll. By default no maximum is set. Can be used to set a limit of e.g. 1000 to avoid when the server that there are thousands of files. Set a value of 0 or negative to allow it. See more details at Batch Consumer.</p> <p>Notice: If this option is in use then the File and FTP components will not perform any sorting. For example if you have 1000000 files and use maxMessagesPerPoll=500, then only the first 500 files will be picked and sorted. You can use the eagerMaxMessagesPerPoll option and set it to true to allow to scan all files first and then sort afterwards.</p>
eagerMaxMessagesPerPoll	true	<p>Camel 2.9.3: Allows for controlling whether the limit from maxMessagesPerPoll is eager or not. If eager then the limit is during the scanning of files. When eager, Camel would scan all files, and then perform sorting. Setting this option to false means to scan all files first, and then limit the poll. Mind that this requires a high memory usage as all file details are in memory to perform the sorting.</p>
minDepth	0	<p>Camel 2.8: The minimum depth to start processing when recursively traversing a directory. Using minDepth=1 means the base directory. Using minDepth=0 means the first sub directory.</p>
processStrategy	null	<p>This option is supported by FTP consumer from Camel 2.8.2, 2.9.</p> <p>Camel 2.8: The maximum depth to traverse when recursively processing a directory. This option is supported by FTP consumer from Camel 2.8 onward.</p> <p>A pluggable <code>org.apache.camel.component.file.GenericFileProcessStrategy</code> allows you to implement your own readLock option or similar. Can also be used to implement your own processStrategy.</p>

		special conditions must be met before a file can be consumed, such as if the starting directory or file exists. If this option is set then the readLock option does not have to be specified.
startingDirectoryMustExist	false	Camel 2.5: Whether the starting directory must exist. Mind that the autoCreate option is default enabled, which means the starting directory is normally created if it doesn't exist. You can disable autoCreate and enable this option if you want the starting directory must exist. Will throw an exception if the directory does not exist.
pollStrategy	null	A pluggable org.apache.camel.spi.PollingConsumerPollStrategy interface that you can provide your own custom implementation to control error handling during the poll operation but <i>before</i> an Exchange has been created by Camel. In other words the error occurred while the polling was gathering information e.g., access to a file network failed so Camel cannot access the files.
sendEmptyMessageWhenIdle	false	The default implementation will log the caused exception at WARN level and ignore it.
consumer.bridgeErrorHandler	false	Camel 2.9: If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.
scheduledExecutorService	null	Camel 2.10: Allows for bridging the consumer to the Camel routing engine which means any exceptions occurred while trying to pickup files, or that are now being processed as a message and handled by the routing Error Handler, the default consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that by default will be logged at WARN/ERROR level. See the following section for more details: <i>How to use the Camel error handler with exceptions triggered outside the routing engine</i> .
scheduler	null	Camel 2.10: Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool. This option allows you to share a thread pool among multiple file consumers.
backoffMultiplier	0	Camel 2.12: To use a custom scheduler to trigger the consumer to run at regular intervals at Polling Consumer, for example there is a Quartz2, and Spring scheduler that supports CRON expressions.
backoffIdleThreshold	0	Camel 2.12: To let the scheduled polling consumer backoff if there is a number of subsequent idles/errors in a row. The multiplier is then applied to the polls that will be skipped before the next actual attempt is happening. This option is in use then backoffErrorThreshold and/or backoffMultiplier must also be configured.
onCompletionExceptionHandler		For more details see: Polling Consumer .
backoffErrorThreshold	0	Camel 2.12: The number of subsequent error polls (failed due to some reason) that should happen before the backoffMultiplier should kick-in.
probeContentType	false	Camel 2.16: To use a custom org.apache.camel.spi.ExceptionHandler to handle any thrown exceptions that happens during the file on completion where the consumer does either a commit or rollback. The default implementation will log any exception at WARN level and ignore.
extendedAttributes	null	Camel 2.15-2.16.x : the default is true. Camel 2.17: To enable gathering extended file attributes through java.nio.file.attribute classes using Files.getAttribute(ava.nio.file.Path, java.lang.String attribute) or Files.readAttributes(ava.nio.file.Path, java.lang.String attributes) depending on the option value. This option supports a comma delimited list of attributes to collect e.g., basic:creationTime, posix:group or simple wildcard e.g., posix:attribute name is not prefixed, the basic attributes are queried. The result is a header with key CamelFileExtendedAttributes and it is of type Map<String, Object> where the key is the name of the attribute e.g., posix:group and the value is the attribute returned by the call to Files.getAttribute() or Files.readAttributes().

Default behavior for file consumer

- By default the file is **not** locked for the duration of the processing.
- After the route has completed, files are moved into the `target.camel` subdirectory, so that they appear to be deleted.

- The File Consumer will always skip any file whose name starts with a dot, such as .., .camel, .m2 or .groovy.
- Only files (not directories) are matched for valid filename, if options such as: include or exclude are used.

Producer

confluenceTableSmall

Name	Default Value	Description
fileExist	override	<p>What to do if a file already exists with the same name. The following values can be specified:</p> <ul style="list-style-type: none"> override replaces the existing file. Append adds content to the existing file. Fail throws a generic <code>FileOperationException</code> indicating that there is already an existing file. Ignore silently ignores the problem and does not override the existing file, but assumes everything is okay. Move (Camel 2.10.1 onward) requires that the option <code>moveExisting</code> be configured as well. The <code>eagerDeleteTargetFile</code> can be used to control what to do if moving the file, and there already exists a file, otherwise causing the move operation to fail. The Move option will move any existing files, before writing the target file. TryRename (Camel 2.11.1 onward) is only applicable if <code>tempFileName</code> option is in use. This allows to try renaming the file from the temporary name to the actual name, without doing any exists check. This check may be faster on some file systems and especially FTP servers.
tempPrefix	null	This option is used to write the file using a temporary name and then, after the write is complete, rename it to the real name. Can be used to identify files being written to and also avoid consumers (not using exclusive read locks) reading in progress files. Is often used by FTP when uploading big files.
tempFileName	null	Camel 2.1: The same as <code>tempPrefix</code> option but offering a more fine grained control on the naming of the temporary filename as it uses the File Language. Camel 2.10.1: Expression (such as File Language) used to compute file name to use when <code>fileExist=Move</code> is configured. To move files into a backup subdirectory just enter backup.
moveExisting	null	<p>This option only supports the following File Language tokens:</p> <ul style="list-style-type: none"> file:name file:name.ext file:name.noext file:onlyname file:onlyname.noext file:ext file:parent <p>Note: the <code>file:parent</code> token is not supported by the FTP component which can only move files to a directory relative to the <i>current</i> directory.</p>
keepLastModified	false	<p>Camel 2.2: Will keep the last modified timestamp from the source file (if any). Will use the Exchange <code>.FILE_LAST_MODIFIED</code> header to located the timestamp. This header can contain either a <code>java.util.Date</code> or <code>long</code> with the timestamp. If the timestamp exists and the option is enabled it will set this timestamp on the written file.</p> <p>Note: This option only applies to the <code>file</code> producer. It <i>cannot</i> be used with any of the FTP producers.</p>
eagerDeleteTargetFile	true	<p>Camel 2.3: Whether or not to eagerly delete any existing target file. This option only applies when you use <code>fileExists=Override</code> and the <code>tempFileName</code> option as well. You can use this to disable (set it to <code>false</code>) deleting the target file before the temp file is written. For example you may write big files and want the target file to exist while the temp file is being written. This ensures that the target file is only deleted at the very last moment, just before the temp file is being renamed to the target filename.</p> <p>From Camel 2.10.1 onward this option is also used to control whether to delete any existing files when <code>fileExist=Move</code> is enabled, and an existing file exists. If this option <code>copyAndDeleteOnRenameFail=false</code>, then an exception will be thrown if an existing file existed. When <code>true</code> the existing file is deleted before the move operation.</p>
doneFileName	null	Camel 2.6: If provided, then Camel will write a second file (called <i>done file</i>) when the original file has been written. The <i>done file</i> will be empty. This option configures what file name to use. You can either specify a fixed name, or you can use dynamic

		placeholders. The <i>done file</i> will always be written in the same folder as the original file. See writing <i>done file</i> section for examples.
allowNullBody	false	Camel 2.10.1: Used to specify if a null body is allowed during file writing. If set to true then an empty file will be created, when set to false, and attempting to send a null body to the file component, a GenericFilewriteException the a message 'Cannot write null body to file' will be thrown. If fileexist=override the file will be truncated. If fileExist=append the file will remain unchanged.
forcewrites	true	Camel 2.10.5/2.11: Whether to force syncing writes to the file system. You can turn this off if you do not want this level of guarantee, for example if writing to logs / audit logs etc; this would yield better performance.
chmod	null	Camel 2.15.0: Specify the file permissions which is sent by the producer, the chmod value must be between 000 and 777; If there is a leading digit like in 0755 we will ignore it.
chmoddirectory	null	Camel 2.17.0: Specify the directory permissions used when the producer creates missing directories, the chmod value must be between 000 and 777; If there is a leading digit like in 0755 we will ignore it.

Default behavior for file producer

- By default it will override any existing file, if one exist with the same name.

Move and Delete operations

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

Lets illustrate this with an example:
`javafrom("file://inbox?move=.done").to("bean:handleOrder");`
When a file is dropped in the inbox folder, the file consumer notices this and creates a new FileExchange that is routed to the handleOrder bean. The bean then processes the File object. At this point in time the file is still located in the inbox folder. After the bean completes, and thus the route is completed, the file consumer will perform the move operation and move the file to the .done sub-folder.

The move and preMove options are considered as a directory name though if you use an expression such as File Language, or Simple then the result of the expression evaluation is the file name to be used e.g., if you set move=../backup/copy-of-\${file:name}
then that's using the File Language which we use return the file name to be used), which can be either relative or absolute. If relative, the directory is created as a sub-folder from within the folder where the file was consumed.

By default, Camel will move consumed files to the .camel sub-folder relative to the directory where the file was consumed.
If you want to delete the file after processing, the route should be:
`javafrom("file://inbox?delete=true").to("bean:handleOrder");`
We have introduced a **pre** move operation to move files **before** they are processed. This allows you to mark which files have been scanned as they are moved to this sub folder before being processed.
`javafrom("file://inbox?preMove=inprogress").to("bean:handleOrder");`
You can combine the **pre** move and the regular move:
`javafrom("file://inbox?preMove=inprogress&move=.done").to("bean:handleOrder");`
So in this situation, the file is in the inprogress folder when being processed and after it's processed, it's moved to the .done folder.

Fine Grained Control Using The move and preMove Options

The **move** and **preMove** options are Expression-based, so we have the full power of the File Language to do advanced configuration of the directory and name pattern.
Camel will, in fact, internally convert the directory name you enter into a File Language expression. So when we enter move=.done Camel will convert this into: \${file:parent} /.done/\${file:onlyname}. This is only done if Camel detects that you have not provided a \${} in the option value yourself. So when you enter a \${} Camel will **not** convert it and thus you have the full power.

So if we want to move the file into a backup folder with today's date as the pattern, we can do:
`move=backup/${date:now:yyyyMMdd}/${file:name}`

About moveFailed

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

See more examples at File Language

Message Headers

The following headers are supported by this component:

File producer only

confluenceTableSmall

Header	Description
CamelFileName	Specifies the name of the file to write (relative to the endpoint directory). This name can be a String; a String with a File Language or Simple expression; or an Expression object. If it's null then Camel will auto-generate a filename based on the message unique ID.
CamelFileNameProduced	The absolute file path (path + name) for the output file that was written. This header is set by Camel and its purpose is providing end-users with the name of the file that was written.
CamelOverrideFileName	Camel 2.11: Is used for overruling camelFileName header and use the value instead (but only once, as the producer will remove this header after writing the file). The value can be only be a String. Notice that if the option fileName has been configured, then this is still being evaluated.

File consumer only

confluenceTableSmall

Header	Description
CamelFileName	Name of the consumed file as a relative file path with offset from the starting directory configured on the endpoint.
CamelFileNameOnly	Only the file name (the name with no leading paths).
CamelFileAbsolute	A boolean option specifying whether the consumed file denotes an absolute path or not. Should normally be false for relative paths. Absolute paths should normally not be used but we added to the move option to allow moving files to absolute paths. But can be used elsewhere as well.
CamelFileAbsolutePath	The absolute path to the file. For relative files this path holds the relative path instead.
CamelFilePath	The file path. For relative files this is the starting directory + the relative filename. For absolute files this is the absolute path.
CamelFileRelativePath	The relative path.
CamelFileParent	The parent path.
CamelFileLength	A long value containing the file size.
CamelFileLastModified	A Long value containing the last modified timestamp of the file. In Camel 2.10.3 and older the type is date.

Batch Consumer

This component implements the Batch Consumer.

Exchange Properties, file consumer only

As the file consumer implements the `BatchConsumer` it supports batching the files it polls. By batching we mean that Camel will add the following additional properties to the Exchange, so you know the number of files polled, the current index, and whether the batch is already completed.

confluenceTableSmall

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

This allows you for instance to know how many files exist in this batch and for instance let the Aggregator2 aggregate this number of files.

Using charset

Available as of Camel 2.9.3

The charset option allows for configuring an encoding of the files on both the consumer and producer endpoints. For example if you read utf-8 files, and want to convert the files to iso-8859-1, you can do:

```
from("file:inbox?charset=utf-8").to("file:outbox?charset=iso-8859-1")
```

You can also use the `convertBodyTo` in the route. In the example below we have still input files in utf-8 format, but we want to convert the file content to a byte array in iso-8859-1 format. And then let a bean process the data. Before writing the content to

the outbox folder using the current charset.

```
from("file:inbox?charset=utf-8").convertBodyTo(byte[].class, "iso-8859-1").to("bean:myBean").to("file:outbox");
If you omit the charset on the consumer endpoint, then Camel does not know the charset of the file, and would by default use "UTF-8". However you can configure a JVM system property to override and use a different default encoding with the key org.apache.camel.default.charset.
```

In the example below this could be a problem if the files is not in UTF-8 encoding, which would be the default encoding for read the files.

In this example when writing the files, the content has already been converted to a byte array, and thus would write the content directly as is (without any further encodings).

```
from("file:inbox").convertBodyTo(byte[].class, "iso-8859-1").to("bean:myBean").to("file:outbox");
You can also override and control the encoding dynamic when writing files, by setting a property on the exchange with the key Exchange.CHARSET_NAME. For example in the route below we set the property with a value from a message header.
```

```
from("file:inbox").convertBodyTo(byte[].class, "iso-8859-1").to("bean:myBean").setProperty(Exchange.CHARSET_NAME, header("someCharsetHeader")).to("file:outbox");
```

We suggest to keep things simpler, so if you pickup files with the same encoding, and want to write the files in a specific encoding, then favor to use the charset option on the endpoints.

Notice that if you have explicit configured a charset option on the endpoint, then that configuration is used, regardless of the Exchange.CHARSET_NAME property.

If you have some issues then you can enable DEBUG logging on org.apache.camel.component.file, and Camel logs when it reads/write a file using a specific charset.

For example the route below will log the following:

```
from("file:inbox?charset=utf-8").to("file:outbox?charset=iso-8859-1")
```

And the logs:

```
DEBUG GenericFileConverter - Read file /Users/davsclaus/workspace/camel/camel-core/target/charset/input/input.txt with charset utf-8 DEBUG FileOperations - Using Reader to write file: target/charset/output.txt with charset: iso-8859-1
```

Common gotchas with folder and filenames

When Camel is producing files (writing files) there are a few gotchas affecting 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. If such a filename is not desired, then you must provide a filename in the CamelFileName message header. The constant, Exchange.FILE_NAME, can also be used.

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

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

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

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

... the same as above, but with CamelFileName:

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

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

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

Filename Expression

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

Consuming files from folders where others drop files directly

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

Using 'done' Files

Available as of Camel 2.6

See also section *writing done files* below.

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

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

Will only consume files from the bar folder, if a done file exists in the same directory as the target files. Camel will automatically delete the *done file* when it's done consuming the files. From Camel 2.9.3 onward Camel will not automatically delete the *done file* if noop=true is configured.

However it is more common to have one *done file* per target file. This means there is a 1:1 correlation. To do this you must use dynamic placeholders in the doneFileName option. Currently Camel supports the following two dynamic tokens: file:name and

`file:name.noext` which must be enclosed in `{}$`. The consumer only supports the static part of the `done file` name as either prefix or suffix (not both).

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

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

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

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

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

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

Writing 'done' Files

Available as of Camel 2.6

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

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

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

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

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

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

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

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

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

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

Examples

Read from a directory and write to another directory

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

Read from a directory and write to another directory using a overrule dynamic name

```
javafrom("file://inputdir/?delete=true").to("file://outputdir?overruleFile=copy-of-${file:name}")
```

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

Reading recursively from a directory and writing to another

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

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

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

Will result in the following output layout:

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

Using flatten

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

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

Will result in the following output layout:

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

Reading from a directory and the default move operation

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

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

Affects the layout as follows:

before

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

after

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

Read from a directory and process the message in java

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

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

Writing to files

Camel is of course also able to write files, i.e. produce files. In the sample below we receive some reports on the SEDA queue that we process before they are being written to a directory.{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/ToFileRouteTest.java}

Write to subdirectory using Exchange.FILE_NAME

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

```
xml <route> <from uri="bean:myBean"/> <to uri="file:/rootDirectory"/> </route>
You can have myBean set the header Exchange.FILE_NAME to values such as:
Exchange.FILE_NAME = hello.txt => /rootDirectory/hello.txt Exchange.FILE_NAME = foo/bye.txt => /rootDirectory/foo/bye.txt
This allows you to have a single route to write files to multiple destinations.
```

Writing file through the temporary directory relative to the final destination

Sometime you need to temporarily write the files to some directory relative to the destination directory. Such situation usually happens when some external process with limited filtering capabilities is reading from the directory you are writing to. In the example below files will be written to the `/var/myapp/filesInProgress` directory and after data transfer is done, they will be atomically moved to the `/var/myapp/finalDirectory` directory.

```
javafrom("direct:start") .to("file:///var/myapp/finalDirectory?tempPrefix=../filesInProgress/");
```

Using Expressions for Filenames

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

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

See File Language for more samples.

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

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

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

Camel uses the absolute file name as the idempotent key, to detect duplicate files. From **Camel 2.11** onward you can customize this key by using an expression in the `idempotentKey` option. For example to use both the name and the file size as the key

```
xml <route> <from uri="file://inbox?idempotent=true&idempotentKey=${file:name}-${file:size}"/> <to
```

```
uri="bean:processInbox"/> </route>
```

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

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

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

Camel will log at DEBUG level if it skips a file because it has been consumed before:

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

Using a file based idempotent repository

In this section we will use the file based idempotent repository

```
org.apache.camel.processor.idempotent.FileIdempotentRepository instead of the in-memory based that is used as default.
```

This repository uses a 1st level cache to avoid reading the file repository. It will only use the file repository to store the content of the 1st level cache. Thereby the repository can survive server restarts. It will load the content of the file into the 1st level cache upon startup. The file structure is very simple as it stores the key in separate lines in the file. By default, the file store has a size limit of 1mb. When the file grows larger Camel will truncate the file store, rebuilding the content by flushing the 1st level cache into a fresh empty file.

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

```
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/idempotent/fileConsumerIdempotentTest.xml}
```

Using a JPA based idempotent repository

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

First we need a persistence-unit in `META-INF/persistence.xml` **where we need to use the class** `org.apache.camel.processor.idempotent.jpa.MessageProcessed` as model.
`{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-jpa/src/test/resources/META-INF/persistence.xml}` Next, we can create our JPA idempotent repository in the spring XML file as well:
`{snippet:id=jpaStore|lang=xml|url=camel/trunk/components/camel-jpa/src/test/resources/org/apache/camel/processor/jpa/fileConsumerJpaldempotentTest-config.xml}` And yes then we just need to refer to the `jpaStore` bean in the file consumer endpoint using the `idempotentRepository` using the `#` syntax option:
`xml <route> <from uri="file://inbox?idempotent=true&idempotentRepository=#jpaStore"/> <to uri="bean:processInbox"/> </route>`

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

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

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

`{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileConsumerFileFilterTest.java}` And then we can configure our route using the `filter` attribute to reference our filter (using `#` notation) that we have defined in the spring XML file:
`xml <!-- define our filter as a plain spring bean --> <bean id="myFilter" class="com.mycompany.MyFileFilter"/> <route> <from uri="file://inbox?filter=#myFilter"/> <to uri="bean:processInbox"/> </route>`

Filtering using ANT path matcher

New options from Camel 2.10 onwards

There are now `antInclude` and `antExclude` options to make it easy to specify ANT style include/exclude without having to define the filter. See the URI options above for more information.

The ANT path matcher is shipped out-of-the-box in the `camel-spring` jar. So you need to depend on `camel-spring` if you are using Maven.

The reasons is that we leverage Spring's AntPathMatcher to do the actual matching.

The file paths is matched with the following rules:

- `?` matches one character
- `*` matches zero or more characters
- `**` matches zero or more directories in a path

The sample below demonstrates how to use it:
`{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/file/SpringFileAntPathMatcherFileFilterTest-context.xml}`

Sorting using Comparator

Camel supports pluggable sorting strategies. This strategy it to use the build in `java.util.Comparator` in Java. You can then configure the endpoint with such a comparator and have Camel sort the files before being processed.

In the sample we have built our own comparator that just sorts by file name:
`{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileSorterRefTest.java}` And then we can configure our route using the `sorter` option to reference to our sorter (`mySorter`) we have defined in the spring XML file:
`xml <!-- define our sorter as a plain spring bean --> <bean id="mySorter" class="com.mycompany.MyFileSorter"/> <route> <from uri="file://inbox?sorter=#mySorter"/> <to uri="bean:processInbox"/> </route>` URI options can reference beans using the `#` syntax
 In the Spring DSL route above notice that we can refer to beans in the Registry by prefixing the id with `#`. So writing `sorter=#mySorter`, will instruct Camel to go look in the Registry for a bean with the ID, `mySorter`.

Sorting using sortBy

Camel supports pluggable sorting strategies. This strategy it to use the File Language to configure the sorting. The `sortBy` option is configured as follows:

`sortBy=group 1;group 2;group 3;...`

Where each group is separated with semi colon. In the simple situations you just use one group, so a simple example could be:
`sortBy=file:name`

This will sort by file name, you can reverse the order by prefixing `reverse:` to the group, so the sorting is now Z..A:
`sortBy=reverse:file:name`

As we have the full power of File Language we can use some of the other parameters, so if we want to sort by file size we do:
`sortBy=file:length`

You can configure to ignore the case, using `ignoreCase:` for string comparison, so if you want to use file name sorting but to ignore the case then we do:
`sortBy=ignoreCase:file:name`

You can combine ignore case and reverse, however reverse must be specified first:

```
sortBy=reverse:ignoreCase:file:name
```

In the sample below we want to sort by last modified file, so we do:

```
sortBy=file:modified
```

And then we want to group by name as a 2nd option so files with same modification is sorted by name:

```
sortBy=file:modified:file:name
```

Now there is an issue here, can you spot it? Well the modified timestamp of the file is too fine as it will be in milliseconds, but what if we want to sort by date only and then subgroup by name?

Well as we have the true power of File Language we can use its date command that supports patterns. So this can be solved as:

```
sortBy=date:file:yyyyMMdd:file:name
```

Yeah, that is pretty powerful, oh by the way you can also use reverse per group, so we could reverse the file names:

```
sortBy=date:file:yyyyMMdd;reverse:file:name
```

Using GenericFileProcessStrategy

The option `processStrategy` can be used to use a custom `GenericFileProcessStrategy` that allows you to implement your own `begin`, `commit` and `rollback` logic.

For instance lets assume a system writes a file in a folder you should consume. But you should not start consuming the file before another `ready` file has been written as well.

So by implementing our own `GenericFileProcessStrategy` we can implement this as:

- In the `begin()` method we can test whether the special `ready` file exists. The `begin` method returns a `boolean` to indicate if we can consume the file or not.
- In the `abort()` method (**Camel 2.10**) special logic can be executed in case the `begin` operation returned `false`, for example to cleanup resources etc.
- In the `commit()` method we can move the actual file and also delete the `ready` file.

Using filter

The `filter` option allows you to implement a custom filter in Java code by implementing the `org.apache.camel.component.file.GenericFileFilter` interface. This interface has an `accept` method that returns a `boolean`. Return `true` to include the file, and `false` to skip the file. From Camel 2.10 onward, there is a `isDirectory` method on `GenericFile` whether the file is a directory. This allows you to filter unwanted directories, to avoid traversing down unwanted directories.

For example to skip any directories which starts with "skip" in the name, can be implemented as follows:

```
{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileConsumerDirectoryFilterTest.java}
```

How to use the Camel error handler to deal with exceptions triggered outside the routing engine

The file and ftp consumers, will by default try to pickup files. Only if that is successful then a Camel Exchange can be created and passed in the Camel routing engine. When the Exchange is processed by the routing engine, then the Camel Error Handling takes over e.g., the `onException` / `errorHandler` in the routes. However outside the scope of the routing engine, any exceptions handling is component specific. Camel offers a `org.apache.camel.spi.ExceptionHandler` that allows components to use that as a pluggable hook for end users to use their own implementation. Camel offers a default `LoggingExceptionHandler` that will log the exception at `ERROR/WARN` level.

For the file and ftp components this would be the case. However if you want to bridge the `ExceptionHandler` so it uses the Camel Error Handling, then you need to implement a custom `ExceptionHandler` that will handle the exception by creating a Camel Exchange and send it to the routing engine; then the error handling of the routing engine can get triggered.

Easier with Camel 2.10

The new option `consumer.bridgeErrorHandler` can be set to `true`, to make this even easier. See further below for more details. Here is such an example based upon an unit test.

First we have a custom `ExceptionHandler` where you can see we deal with the exception by sending it to a Camel Endpoint named `direct:file-error:{snippet:id=e1|title=MyExceptionHandler|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileConsumerCustomExceptionHandlerTest.java}`

Then we have a Camel route that uses the Camel routing error handler, which is the `onException` where we handle any `IOException` being thrown. We then send the message to the same `direct:file-error` endpoint, where we handle it by transforming it to a message, and then being sent to a Mock endpoint. This is just for testing purpose. You can handle the exception in any custom way you want, such as using a Bean or sending an email, etc.

Notice how we configure our custom `MyExceptionHandler` by using the `consumer.exceptionHandler` option to refer to `#myExceptionHandler` which is a id of the bean registered in the Registry. If using Spring XML or OSGi Blueprint, then that would be a `<bean id="myExceptionHandler" class="com.foo.MyExceptionHandler"/>:{snippet:id=e2|title=Camel route with routing engine error handling|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileConsumerCustomExceptionHandlerTest.java}`

The source code for this example can be seen here

Using `consumer.bridgeErrorHandler`

Available as of Camel 2.10

If you want to use the Camel Error Handler to deal with any exception occurring in the file consumer, then you can enable the `consumer.bridgeErrorHandler` option as shown below:{snippet:id=e2|title=Using consumer.bridgeErrorHandler|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileConsumerBridgeRouteExceptionHandlerTest.java}So all you have to do is to enable this option, and the error handler in the route will take it from there.

Important when using `consumer.bridgeErrorHandler`

When using `consumer.bridgeErrorHandler`, then interceptors, `OnCompletions` does **not** apply. The Exchange is processed directly by the Camel Error Handler, and does not allow prior actions such as interceptors, `onCompletion` to take action.

Debug logging

This component has log level TRACE that can be helpful if you have problems.

Endpoint See Also

- File Language
- FTP
- Polling Consumer

FLATPACK COMPONENT

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

Notice: This component only supports consuming from flatpack files to Object model. You can not (yet) write from Object model to flatpack format.

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

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

URI format

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

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

```
flatpack:someName[?options]
```

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

URI Options

Name	Default Value	Description
<code>delimiter</code>	<code>,</code>	The default character delimiter for delimited files.
<code>textQualifier</code>	<code>"</code>	The text qualifier for delimited files.
<code>ignoreFirstRecord</code>	<code>true</code>	Whether the first line is ignored for delimited files (for the column headers).
<code>splittows</code>	<code>true</code>	The component can either process each row one by one or the entire content at once.
<code>allowExtraLines</code>	<code>false</code>	Camel 2.9.7 and 2.10.5 onwards: Allows for lines to be longer than expected and ignores the extra characters.
<code>ignoreExtraLines</code>	<code>false</code>	Camel 2.9.7 and 2.10.5 onwards: Allows for lines to be longer than expected and ignores the extra characters.

Examples

- `flatpack:fixed:foo.pzmap.xml` creates a fixed-width endpoint using the `foo.pzmap.xml` file configuration.
- `flatpack:delim:bar.pzmap.xml` creates a delimited endpoint using the `bar.pzmap.xml` file configuration.
- `flatpack:foo` creates a delimited endpoint called `foo` with no file configuration.

Message Headers

Camel will store the following headers on the IN message:

Header	Description
<code>camelFlatpackCounter</code>	The current row index. For <code>splittows=false</code> the counter is the total number of rows.

Message Body

The component delivers the data in the IN message as a `org.apache.camel.component.flatpack.DataSetList` object that has converters for `java.util.Map` or `java.util.List`. Usually you want the Map if you process one row at a time (`splitRows=true`). Use `List` for the entire content (`splitRows=false`), where each element in the list is a Map. Each Map contains the key for the column name and its corresponding value.

For example to get the `firstname` from the sample below:

```
Map row = exchange.getIn().getBody(Map.class);
String firstName = row.get("FIRSTNAME");
```

However, you can also always get it as a `List` (even for `splitRows=true`). The same example:

```
List data = exchange.getIn().getBody(List.class);
Map row = (Map)data.get(0);
String firstName = row.get("FIRSTNAME");
```

Header and Trailer records

The header and trailer notions in Flatpack are supported. However, you **must** use fixed record IDs:

- header for the header record (must be lowercase)
- trailer for the trailer record (must be lowercase)

The example below illustrates this fact that we have a header and a trailer. You can omit one or both of them if not needed.

```
<RECORD id="header" startPosition="1" endPosition="3" indicator="HBT">
  <COLUMN name="INDICATOR" length="3"/>
  <COLUMN name="DATE" length="8"/>
</RECORD>

<COLUMN name="FIRSTNAME" length="35" />
<COLUMN name="LASTNAME" length="35" />
<COLUMN name="ADDRESS" length="100" />
<COLUMN name="CITY" length="100" />
<COLUMN name="STATE" length="2" />
<COLUMN name="ZIP" length="5" />

<RECORD id="trailer" startPosition="1" endPosition="3" indicator="FBT">
  <COLUMN name="INDICATOR" length="3"/>
  <COLUMN name="STATUS" length="7"/>
</RECORD>
```

Using the endpoint

A common use case is sending a file to this endpoint for further processing in a separate route. For example:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="file:/someDirectory"/>
    <to uri="flatpack:foo"/>
  </route>

  <route>
    <from uri="flatpack:foo"/>
    ...
  </route>
</camelContext>
```

You can also convert the payload of each message created to a `Map` for easy Bean Integration

FLATPACK DATAFORMAT

The Flatpack component ships with the Flatpack data format that can be used to format between fixed width or delimited text messages to a `List` of rows as `Map`.

- marshal = from `List<Map<String, Object>>` to `OutputStream` (can be converted to `String`)
 - unmarshal = from `java.io.InputStream` (such as a `File` or `String`) to a `java.util.List` as an `org.apache.camel.component.flatpack.DataSetList` instance.
- The result of the operation will contain all the data. If you need to process each row one by one you can split the exchange, using Splitter.

Notice: The Flatpack library does currently not support header and trailers for the marshal operation.

Options

The data format has the following options:

Option	Default	Description
--------	---------	-------------

definition	<code>null</code>	The flatpack pzmap configuration file. Can be omitted in simpler situations, but its preferred to use the pzmap.
fixed	<code>false</code>	Delimited or fixed.
ignoreFirstRecord	<code>true</code>	Whether the first line is ignored for delimited files (for the column headers).
textQualifier	<code>"</code>	If the text is qualified with a char such as <code>"</code> .
delimiter	<code>,</code>	The delimiter char (could be <code>;</code> , <code>,</code> or similar)
parserFactory	<code>null</code>	Uses the default Flatpack parser factory.
allowShortLines	<code>false</code>	Camel 2.9.7 and 2.10.5 onwards: Allows for lines to be shorter than expected and ignores the extra characters.
ignoreExtraColumns	<code>false</code>	Camel 2.9.7 and 2.10.5 onwards: Allows for lines to be longer than expected and ignores the extra characters.

Usage

To use the data format, simply instantiate an instance and invoke the marshal or unmarshal operation in the route builder:

```
FlatpackDataFormat fp = new FlatpackDataFormat();
fp.setDefinition(new ClassPathResource("INVENTORY-Delimited.pzmap.xml"));
...
from("file:order/in").unmarshal(df).to("seda:queue:neworder");
```

The sample above will read files from the `order/in` folder and unmarshal the input using the Flatpack configuration file `INVENTORY-Delimited.pzmap.xml` that configures the structure of the files. The result is a `datasetList` object we store on the SEDA queue.

```
FlatpackDataFormat df = new FlatpackDataFormat();
df.setDefinition(new ClassPathResource("PEOPLE-FixedLength.pzmap.xml"));
df.setFixed(true);
df.setIgnoreFirstRecord(false);
from("seda:people").marshal(df).convertBodyTo(String.class).to("jms:queue:people");
```

In the code above we marshal the data from a Object representation as a `List` of rows as Maps. The rows as Map contains the column name as the key, and the the corresponding value. This structure can be created in Java code from e.g. a processor. We marshal the data according to the Flatpack format and convert the result as a `String` object and store it on a JMS queue.

Dependencies

To use Flatpack in your camel routes you need to add the a dependency on **camel-flatpack** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-flatpack</artifactId>
<version>x.x.x</version>
</dependency>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

FREEMARKER

The **freemarker**: component allows for processing a message using a FreeMarker template. This can be ideal when using Templating to generate responses for requests.

Maven users will need to add the following dependency to their pom.xml for this component:
`xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-freemarker</artifactId> <version>x.x.x</version> <!-- use the same version as your Camel core version --> </dependency>`

URI format

`freemarker:templateName[?options]`

Where `templateName` is the classpath-local URI of the template to invoke; or the complete URL of the remote template (eg: `file://folder/myfile.ftl`).

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

Options

confluenceTableSmall

Option	Default	Description
contentCache	true	Cache for the resource content when it's loaded. Note: as of Camel 2.9 cached resource content can be cleared via JMX using the endpoint's clearContentCache operation.
encoding	null	Character encoding of the resource content.
templateUpdateDelay	5	Camel 2.9: Number of seconds the loaded template resource will remain in the cache.

Headers

Headers set during the FreeMarker evaluation are returned to the message and added as headers. This provides a mechanism for the FreeMarker component to return values to the Message.

An example: Set the header value of fruit in the FreeMarker template:

```
 ${request.setHeader('fruit', 'Apple')}
```

The header, fruit, is now accessible from the message.out.headers.

FreeMarker Context

Camel will provide exchange information in the FreeMarker context (just a Map). The Exchange is transferred as:
confluenceTableSmall

key	value
exchange	The Exchange itself.
exchange.properties	The Exchange properties.
headers	The headers of the In message.
camelContext	The Camel Context.
request	The In message.
body	The In message body.
response	The Out message (only forInOut message exchange pattern).

From Camel 2.14, you can setup your custom FreeMarker context in the message header with the key

"**CamelFreemarkerDataModel**" just like this

```
Map<String, Object> variableMap = new HashMap<String, Object>(); variableMap.put("headers", headersMap);  
variableMap.put("body", "Monday"); variableMap.put("exchange", exchange);  
exchange.getIn().setHeader("CamelFreemarkerDataModel", variableMap);
```

Hot reloading

The FreeMarker template resource is by default **not** hot reloadable for both file and classpath resources (expanded jar). If you set contentCache=false, then Camel will not cache the resource and hot reloading is thus enabled. This scenario can be used in development.

Dynamic templates

Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

confluenceTableSmall

Header	Type	Description	Support Version
FreemarkerConstants.FREEMARKER_RESOURCE	org.springframework.core.io.Resource	The template resource	<= 2.1
FreemarkerConstants.FREEMARKER_RESOURCE_URI	String	A URI for the template resource to use instead of the endpoint configured.	>= 2.1
FreemarkerConstants.FREEMARKER_TEMPLATE	String	The template	>= 2.1

to use instead
of the endpoint
configured.

Samples

For example you could use something like:

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

To use a FreeMarker 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("freemarker:com/acme/MyResponse.ftl").to("activemq:Another.Queue");

And to disable the content cache, e.g. for development usage where the .ftl template should be hot reloaded:

```
from("activemq:My.Queue").to("freemarker:com/acme/MyResponse.ftl?contentCache=false").to("activemq:Another.Queue");
```

And a file-based resource:

```
from("activemq:My.Queue").to("freemarker:file://myfolder/MyResponse.ftl?contentCache=false").to("activemq:Another.Queue");
```

In Camel 2.1 it's possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").setHeader(FreemarkerConstants.FREEMARKER_RESOURCE_URI).constant("path/to/my/template.ftl").to("freemarker:dummy");
```

The Email Sample

In this sample we want to use FreeMarker templating for an order confirmation email. The email template is laid out in FreeMarker as:

```
Dear ${headers.lastName}, ${headers.firstName} Thanks for the order of ${headers.item}. Regards Camel Riders Bookstore  
$body
```

And the java code:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-freemarker/src/test/java/org/apache/camel/component/freemarker/FreemarkerLetterTest.java}
```

Endpoint See Also

FTP/SFTP/FTPS COMPONENT

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

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

```
xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-ftp</artifactId> <version>x.x.x</version>See the  
documentation of the Apache Commons <!-- use the same version as your Camel core version --> </dependency>
```

More options

See File for more options as all the options from File is inherited.

Absolute paths

Absolute path is **not** supported.

Camel 2.16 will translate absolute paths to relative ones by trimming all leading slashes from directoryname. There'll be WARN message printed in the logs.

Consuming from remote FTP server

Make sure you read the section titled *Default when consuming files* further below for details related to consuming files.

URI format

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

ftps://[username@]hostname[:port]/directoryname[?options]

Where **directoryname** represents the underlying directory. The directory name is a relative path. Absolute paths are **not** supported. The relative path can contain nested folders, such as /inbox/us.

For Camel versions before **Camel 2.16**, the directoryName **must** exist already as this component does not support the autoCreate option (which the file component does). The reason is that its the FTP administrator (FTP server) task to properly setup user accounts, and home directories with the right file permissions etc.

For **Camel 2.16**, autoCreate option is supported. When consumer starts, before polling is scheduled, there's additional FTP operation performed to create the directory configured for endpoint. The default value for autoCreate is true.

If no **username** is provided, then anonymous login is attempted using no password.

If no **port** number is provided, Camel will provide default values according to the protocol (ftp = 21, sftp = 22, ftps = 2222).

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

This component uses two different libraries for the actual FTP work. FTP and FTPS uses Apache Commons Net while SFTP uses JCraft JSCH.

The FTPS component is only available in Camel 2.2 or newer.

FTPS (also known as FTP Secure) is an extension to FTP that adds support for the Transport Layer Security (TLS) and the

Secure Sockets Layer (SSL) cryptographic protocols.

URI Options

The options below are exclusive for the FTP component.

More options

See File for more options as all the options from File is inherited.

confluenceTableSmall

Name	Default Value	Description
username	null	Specifies the username to use to log in to the remote file system.
password	null	Specifies the password to use to log in to the remote file system.
account	null	Camel 2.15.2: Specified the account to use to login to the remote FTP site.
binary	false	Specifies the file transfer mode, BINARY or ASCII. Default is ASCII (false).
disconnect	false	Camel 2.2: Whether or not to disconnect from remote FTP server right after producer. Disconnect will only disconnect the current connection to which you want to stop, then you need to stop the consumer/route instead.
localWorkDirectory	null	When consuming, a local work directory can be used to store the remote loading the content into memory. This is beneficial, if you consume a very large file. See below for more details.
passiveMode	false	FTP and FTPS only: Specifies whether to use passive mode connection.
securityProtocol	TLS	FTPS only: Sets the underlying security protocol. The following values are supported: TLS: Transport Layer Security SSL: Secure Sockets Layer
disableSecureDataChannelDefaults	false	Camel 2.4: FTPS only: Whether or not to disable using default values for secure data transfer. You can set this option to true if you want to be in control of execProt should be used.
download	true	Camel 2.11: Whether the FTP consumer should download the file. If this option is false, the consumer will still trigger a Camel Exchange if it finds a file name, file size, etc. It's just that the file will not be downloaded.
streamDownload	false	Camel 2.11: Whether the consumer should download the entire file up front or pass an InputStream read from the remote resource rather than an in-memory Camel Exchange. This option is ignored if download is false or is local.
execProt	null	Camel 2.4: FTPS only: Will by default use option P if secure data channel values are: C: Clear S: Safe (SSL protocol only) E: Confidential (SSL protocol only) P: Private
execPbsz	null	Camel 2.4: FTPS only: This option specifies the buffer size of the secure data channel. It is only used if useSecureDataChannel has been enabled and this option has not been specified.
isImplicit	false	FTPS only: Sets the security mode(implicit/explicit). Default is explicit (false).
knownHostsFile	null	SFTP only: Sets the known_hosts file, so that the SFTP endpoint can do private key verification.
useUserKnownHostsFile	true	SFTP only: Camel 2.18: If knownHostFile has not been explicitly configured, then System.getProperty(user.home).ssh/known_hosts is used.
knownHostsUri	null	SFTP only: Camel 2.11.1: Sets the known_hosts file (loaded from classpath) so that the SFTP endpoint can do host key verification.
keyPair	null	SFTP only: Camel 2.12.0: Sets the Java KeyPair for SSH public key authentication.
privatekeyFile	null	SFTP only: Set the private key file to that the SFTP endpoint can do private key verification.
privatekeyuri	null	SFTP only: Camel 2.11.1: Set the private key file (loaded from classpath) so that the SFTP endpoint can do private key verification.
privatekey	null	SFTP only: Camel 2.11.1: Set the private key as byte[] to that the SFTP endpoint can do private key verification.
privatekeyfilepassphrase	null	SFTP only: Deprecated: use privateKeyPassphrase instead. Set the private key file passphrase to that the SFTP endpoint can do private key verification.
privatekeypassphrase	null	SFTP only: Camel 2.11.1: Set the private key file passphrase to that the SFTP endpoint can do private key verification.
preferredAuthentications	null	SFTP only: Camel 2.10.7, 2.11.2, 2.12.0: set the preferred authentication mechanism. example include:password,publickey. If not specified the default list from SFTP client is used.
ciphers	null	Camel 2.8.2, 2.9: SFTP only Set a comma separated list of ciphers that the SFTP client supports.

		Possible cipher names are defined by JCraft JSCH. Some examples incl cbc, blowfish-cbc, aes192-cbc, aes256-cbc. If not specified the default list
fastExistsCheck	false	Camel 2.8.2, 2.9: If set this option to be true, camel-ftp will use the list file some FTP server may not support to list the file directly, if the option is false the directory and check if the file exists. Note from Camel 2.10.1 onward readLock=changed to control whether it performs a fast check to update speed up the process if the FTP server has a lot of files.
strictHostKeyChecking	no	SFTP only: Camel 2.2: Sets whether to use strict host key checking. Po does not make sense to use as Camel cannot answer the question for you. Note: The default in Camel 2.1 and below was ask.
maximumReconnectAttempts	3	Specifies the maximum reconnect attempts Camel performs when it tries 0 to disable this behavior.
reconnectDelay	1000	Delay in millis Camel will wait before performing a reconnect attempt.
connectTimeout	10000	Camel 2.4: Is the connect timeout in millis. This corresponds to using ft FTP/FTPS. For SFTP this option is also used when attempting to connect.
soTimeout	null / 30000	FTP and FTPS Only: Camel 2.4: Is the socketOptions.SO_TIMEOUT value this to a value such as 300000 (5 minutes) to not hang a connection. On JSCH Session instance. Also SFTP from Camel 2.14.3/2.15.3/2.16 onwards. From Camel 2.16 onwards the default is 300000 (300 sec).
timeout	30000	FTP and FTPS Only: Camel 2.4: Is the data timeout in millis. This corre for the FTP/FTPS. For SFTP there is no data timeout.
throwExceptionOnConnectFailed	false	Camel 2.5: Whether or not to throw an exception if a successful connect allows a custom pollStrategy to deal with the exception, for example to
siteCommand	null	FTP and FTPS Only: Camel 2.5: To execute site commands after success separated using a new line character (\n). Use help site to see which s
stepwise	true	Camel 2.6: Whether or not stepwise traversing directories should be use one directory at a time. See more details below. You can disable this in c
separator	UNIX	Camel 2.6: Dictates what path separator char to use when uploading file altering it. UNIX = Use unix style path separators, windows = Use Windows Since Camel 2.15.2: The default value is changed to UNIX style path, Auto.
chmod	null	SFTP Producer Only: Camel 2.9: Allows you to set chmod on the store
compression	0	SFTP Only: Camel 2.8.3/2.9: To use compression. Specify a level from add the needed JSCH zlib JAR to the classpath for compression support
receiveBufferSize	32768	FTP/FTPS Only: Camel 2.15.1: The buffer size for downloading files. Tr
ftpClient	null	FTP and FTPS Only: Camel 2.1: Allows you to use a custom org.apache.camel.impl.FtpClient instance.
ftpClientConfig	null	FTP and FTPS Only: Camel 2.1: Allows you to use a custom org.apache.camel.impl.FtpClientConfig instance.
ftpClientConfig.***		FTP and FTPS Only: To configure various options on the FtpClient instance. ftpClientConfig.receiveDataSocketBufferSize=8192&ftpClientConfig.***
serverAliveInterval	0	SFTP Only: Camel 2.8: Allows you to set the serverAliveInterval of the session.
serverAliveCountMax	1	SFTP Only: Camel 2.8: Allows you to set the serverAliveCountMax of the session.
ftpClient.trustStore.file	null	FTPS Only: Sets the trust store file, so that the FTPS client can look up the trust store.
ftpClient.trustStore.type	JKS	FTPS Only: Sets the trust store type.
ftpClient.trustStore.algorithm	sunX509	FTPS Only: Sets the trust store algorithm.
ftpClient.trustStore.password	null	FTPS Only: Sets the trust store password.
ftpClient.keyStore.file	null	FTPS Only: Sets the key store file, so that the FTPS client can look up the key store.
ftpClient.keyStore.type	JKS	FTPS Only: Sets the key store type.
ftpClient.keyStore.algorithm	sunX509	FTPS Only: Sets the key store algorithm.
ftpClient.keyStore.password	null	FTPS Only: Sets the key store password.
ftpClient.keyStore.keyPassword	null	FTPS Only: Sets the private key password.
sslContextParameters	null	FTPS Only: Camel 2.9: Reference to a org.apache.camel.util.jsse.SslContextParameters instance. This reference overrides any configured SSL related options on FtpClient etc.) set on FtpsConfiguration. See Using the JSSE Configuration Utility
proxy	null	SFTP Only: Camel 2.10.7, 2.11.1: Reference to a com.jcraft.jsch.Proxy instance. consume/send messages from the target SFTP host.
useList	true	

		FTP/FTPS Only: Camel 2.12.1: Whether the consumer should use FTP to see which files exists. If this option is set to false, then stepwise=false must be configured to a fixed name, so the consumer knows to this only that single file can be retrieved. See further below for more details.
ignoreFileNotFoundExceptionOrPermissionError	false	Camel 2.12.1: Whether the consumer should ignore when a file was absent (for some reason), or failure due insufficient file permission error. Camel 2.14.1: as well.
sendNoop	true	Camel 2.16: Producer only. Whether to send a noop command as a pre-FTP server. This is enabled by default as a validation of the connection: connect to be able to upload the file. However if this causes problems, you can turn it off.
jschLogLevel	WARN	SFTP Only: Camel 2.15.3/2.16: The logging level to use for JSCH activity. The default at INFO level the threshold is WARN by default.
bulkRequest		SFTP Only: Camel 2.17.1: Specifies how many requests may be outstanding. This value may slightly improve file transfer speed but will increase memory usage.
disconnectOnBatchComplete	false	Camel 2.18: Whether or not to disconnect from remote FTP server after consumer and producer. Disconnect will only disconnect the current consumer which you want to stop, then you need to stop the consumer.
activePortRange		Camel 2.18: Set the client side port range in active mode. The syntax is: inclusive, eg 10000-19999 to include all 1xxxx ports.

FTPS component default trust store

When using the `ftpClient`. properties related to SSL with the FTPS component, the trust store accepts all certificates. If you only want trust selective certificates, you have to configure the trust store with the `ftpClient.trustStore.xxx` options or by configuring a custom `ftpClient`.

When using `sslContextParameters`, the trust store is managed by the configuration of the provided `SSLContextParameters` instance.

You can configure additional options on the `ftpClient` and `ftpClientConfig` from the URI directly by using the `ftpClient.` or `ftpClientConfig.` prefix.

For example to set the `setdataTimeout` on the `FTPClient` to 30 seconds you can do:

```
from("ftp://foo@myserver?password=secret&ftpClient.dataTimeout=30000").to("bean:foo");
```

You can mix and match and have use both prefixes, for example to configure date format or timezones.

```
from("ftp://foo@myserver?
```

```
password=secret&ftpClient.dataTimeout=30000&ftpClientConfig.serverLanguageCode=fr").to("bean:foo");
```

You can have as many of these options as you like.

See the documentation of the Apache Commons FTP `FTPClientConfig` for possible options and more details. And as well for Apache Commons FTP `FTPClient`.

If you do not like having many and long configuration in the url you can refer to the `ftpClient` or `ftpClientConfig` to use by letting Camel lookup in the Registry for it.

For example:

```
<bean id="myConfig" class="org.apache.commons.net.ftp.FTPClientConfig"> <property name="lenientFutureDates" value="true"/> <property name="serverLanguageCode" value="fr"/> </bean>
```

And then let Camel lookup this bean when you use the # notation in the url.

```
from("ftp://foo@myserver?password=secret&ftpClientConfig=#myConfig").to("bean:foo");
```

More URI options

title:More options

See File2 as all the options there also applies for this component.

Examples

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

FTP Consumer does not support concurrency

The FTP consumer (with the same endpoint) does not support concurrency (the backing FTP client is not thread safe).

You can use multiple FTP consumers to poll from different endpoints. It is only a single endpoint that does not support concurrent consumers.

The FTP producer does **not** have this issue, it supports concurrency.

More information

This component is an extension of the File component. So there are more samples and details on the File component page.

Default when consuming files

The FTP consumer will by default leave the consumed files untouched on the remote FTP server. You have to configure it explicitly if you want it to delete the files or move them to another location. For example you can use `delete=true` to delete the files, or use `move=.done` to move the files into a hidden done sub directory.

The regular File consumer is different as it will by default move files to a .camel sub directory. The reason Camel does **not** do this by default for the FTP consumer is that it may lack permissions by default to be able to move or delete files.

limitations

The option `readLock` can be used to force Camel **not** to consume files that are currently being written. However, this option is turned off by default, as it requires that the user has write access. See the options table at File2 for more details about read locks. There are other solutions to avoid consuming files that are currently being written over FTP; for instance, you can write to a temporary destination and move the file after it has been written.

When moving files using `move` or `premove` option the files are restricted to the `FTP_ROOT` folder. That prevents you from moving files outside the FTP area. If you want to move files to another area you can use soft links and move files into a soft linked folder.

Message Headers

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

Header	Description
<code>CamelFileName</code>	Specifies the output file name (relative to the endpoint directory) to be used for the output message when sending to the endpoint. If this is not present and no expression either, then a generated message ID is used as the filename instead.
<code>CamelFileNameProduced</code>	The actual 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.
<code>CamelBatchIndex</code>	Current index out of total number of files being consumed in this batch.
<code>CamelBatchSize</code>	Total number of files being consumed in this batch.
<code>CamelFileHost</code>	The remote hostname.
<code>CamelFileLocalworkPath</code>	Path to the local work file, if local work directory is used.

In addition the FTP/FTPS consumer and producer will enrich the Camel Message with the following headers
confluenceTableSmall

Header	Description
<code>CamelFtpReplyCode</code>	Camel 2.11.1: The FTP client reply code (the type is a integer)
<code>CamelFtpReplyString</code>	Camel 2.11.1: The FTP client reply string

About timeouts

The two set of libraries (see top) have different APIs for setting timeout. You can use the `connectTimeout` option for both of them to set a timeout in millis to establish a network connection. An individual `sotimeout` can also be set on the FTP/FTPS, which corresponds to using `ftpClient.sotimeout`. Notice SFTP will automatically use `connectTimeout` as its `sotimeout`. The `timeout` option only applies for FTP/FTSP as the data timeout, which corresponds to the `ftpClient.dataTimeout` value. All timeout values are in millis.

Using Local Work Directory

Camel supports consuming from remote FTP servers and downloading the files directly into a local work directory. This avoids reading the entire remote file content into memory as it is streamed directly into the local file using `FileOutputStream`.

Camel will store to a local file with the same name as the remote file, though with `.inprogress` as extension while the file is being downloaded. Afterwards, the file is renamed to remove the `.inprogress` suffix. And finally, when the Exchange is complete the local file is deleted.

So if you want to download files from a remote FTP server and store it as files then you need to route to a file endpoint such as:

```
javafrom("ftp://someone@someserver.com?password=secret&localWorkDirectory=/tmp").to("file://inbox"); Optimization by renaming work file
```

The route above is ultra efficient as it avoids reading the entire file content into memory. It will download the remote file directly to a local file stream. The `java.io.File` handle is then used as the Exchange body. The file producer leverages this fact and can work directly on the work file `java.io.File` handle and perform a `java.io.File.rename` to the target filename. As Camel knows it's a local work file, it can optimize and use a rename instead of a file copy, as the work file is meant to be deleted anyway.

Stepwise changing directories

Camel FTP can operate in two modes in terms of traversing directories when consuming files (eg downloading) or producing files (eg uploading)

- stepwise

- not stepwise

You may want to pick either one depending on your situation and security issues. Some Camel end users can only download files if they use stepwise, while others can only download if they do not. At least you have the choice to pick (from Camel 2.6 onwards).

In Camel 2.0 - 2.5 there is only one mode and it is:

- before 2.5 not stepwise
- 2.5 stepwise

From Camel 2.6 onwards there is now an option `stepwise` you can use to control the behavior.

Note that stepwise changing of directory will in most cases only work when the user is confined to its home directory and when the home directory is reported as "/".

The difference between the two of them is best illustrated with an example. Suppose we have the following directory structure on the remote FTP server we need to traverse and download files:

/one /one/two /one/two/sub-a /one/two/sub-b

And that we have a file in each of sub-a (a.txt) and sub-b (b.txt) folder.

Using `stepwise=true` (default mode)

```
TYPE A 200 Type set to A PWD 257 "/" is current directory. CWD one 250 CWD successful. "/one" is current directory. CWD two 250 CWD successful. "/one/two" is current directory. SYST 215 UNIX emulated by FileZilla PORT 127,0,0,1,17,94 200 Port command successful LIST 150 Opening data channel for directory list. 226 Transfer OK CWD sub-a 250 CWD successful. "/one/two/sub-a" is current directory. PORT 127,0,0,1,17,95 200 Port command successful LIST 150 Opening data channel for directory list. 226 Transfer OK CDUP 200 CDUP successful. "/one/two" is current directory. CWD sub-b 250 CWD successful. "/one/two/sub-b" is current directory. PORT 127,0,0,1,17,96 200 Port command successful LIST 150 Opening data channel for directory list. 226 Transfer OK CDUP 200 CDUP successful. "/one/two" is current directory. CWD / 250 CWD successful. "/" is current directory. PWD 257 "/" is current directory. CWD one 250 CWD successful. "/one" is current directory. CWD two 250 CWD successful. "/one/two" is current directory. PORT 127,0,0,1,17,97 200 Port command successful RETR foo.txt 150 Opening data channel for file transfer. 226 Transfer OK CWD / 250 CWD successful. "/" is current directory. PWD 257 "/" is current directory. CWD one 250 CWD successful. "/one" is current directory. CWD two 250 CWD successful. "/one/two" is current directory. CWD sub-a 250 CWD successful. "/one/two/sub-a" is current directory. PORT 127,0,0,1,17,98 200 Port command successful RETR a.txt 150 Opening data channel for file transfer. 226 Transfer OK CWD / 250 CWD successful. "/" is current directory. PWD 257 "/" is current directory. CWD one 250 CWD successful. "/one" is current directory. CWD two 250 CWD successful. "/one/two" is current directory. CWD sub-b 250 CWD successful. "/one/two/sub-b" is current directory. PORT 127,0,0,1,17,99 200 Port command successful RETR b.txt 150 Opening data channel for file transfer. 226 Transfer OK CWD / 250 CWD successful. "/" is current directory. QUIT 221 Goodbye disconnected.
```

As you can see when stepwise is enabled, it will traverse the directory structure using CD xxx.

Using `stepwise=false`

```
230 Logged on TYPE A 200 Type set to A SYST 215 UNIX emulated by FileZilla PORT 127,0,0,1,4,122 200 Port command successful LIST one/two 150 Opening data channel for directory list 226 Transfer OK PORT 127,0,0,1,4,123 200 Port command successful LIST one/two/sub-a 150 Opening data channel for directory list 226 Transfer OK PORT 127,0,0,1,4,124 200 Port command successful LIST one/two/sub-b 150 Opening data channel for directory list 226 Transfer OK PORT 127,0,0,1,4,125 200 Port command successful RETR one/two/foo.txt 150 Opening data channel for file transfer. 226 Transfer OK PORT 127,0,0,1,4,126 200 Port command successful RETR one/two/sub-a/a.txt 150 Opening data channel for file transfer. 226 Transfer OK PORT 127,0,0,1,4,127 200 Port command successful RETR one/two/sub-b/b.txt 150 Opening data channel for file transfer. 226 Transfer OK QUIT 221 Goodbye disconnected.
```

As you can see when not using stepwise, there are no CD operation invoked at all.

Samples

In the sample below we set up 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.{snippet:id=e1|lang=java|url=camel/trunk/components/camel-ftp/src/test/java/org/apache/camel/component/file/remote/FromFtpToBinarySampleTest.java}And the route using Spring DSL:
`xml <route> <from uri="ftp://scott@localhost/public/reports?password=tiger&binary=true&delay=60000"/> <to uri="file://target/test-reports"/> </route>`

Consuming a remote FTPS server (implicit SSL) and client authentication

```
from("ftps://admin@localhost:2222/public/camel?password=admin&securityProtocol=SSL&isImplicit=true
&ftpClient.keyStore.file=./src/test/resources/server.jks
&ftpClient.keyStore.password=password&ftpClient.keyStore.keyPassword=password").to("bean:foo");
```

Consuming a remote FTPS server (explicit TLS) and a custom trust store configuration

```
from("ftps://admin@localhost:2222/public/camel?
password=admin&ftpClient.trustStore.file=../src/test/resources/server.jks&ftpClient.trustStore.password=password")
.to("bean:foo");
```

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

Camel supports pluggable filtering strategies. This strategy can be provided by implementing `org.apache.camel.component.file.GenericFileFilter` in Java. You can then configure the endpoint with such a filter to skip certain filters before being processed.

In the sample we have built our own filter that only accepts files starting with report in the filename.

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-
ftp/src/test/java/org/apache/camel/component/file/remote/FromFtpRemoteFileFilterTest.java}And then we can configure our route using the filter attribute to reference our filter (using # notation) that we have defined in the spring XML file:  
xml <!-- define our sorter as a plain spring bean --> <bean id="myFilter" class="com.mycompany.MyFileFilter"/> <route> <from uri="ftp://someuser@someftpserver.com?password=secret&filter=#myFilter"/> <to uri="bean:processInbox"/> </route>
```

Filtering using ANT path matcher

The ANT path matcher is a filter that is shipped out-of-the-box in the **camel-spring** jar. So you need to depend on **camel-spring** if you are using Maven.

The reason is that we leverage Spring's AntPathMatcher to do the actual matching.

The file paths are matched with the following rules:

- ? matches one character
- * matches zero or more characters
- ** matches zero or more directories in a path

The sample below demonstrates how to use it:{snippet:id=example|lang=xml|url=camel/trunk/tests/camel-
itest/src/test/resources/org/apache/camel/itest/ftp/SpringFileAntPathMatcherRemoteFileFilterTest-context.xml}

Using a proxy with SFTP

To use an HTTP proxy to connect to your remote host, you can configure your route in the following way:

```
xml<!-- define our sorter as a plain spring bean --> <bean id="proxy" class="com.jcraft.jsch.ProxyHTTP"> <constructor-arg
```

```
value="localhost"/> <constructor-arg value="7777"/> </bean> <route> <from uri="sftp://localhost:9999/root?
```

```
username=admin&password=admin&proxy=#proxy"/> <to uri="bean:processFile"/> </route>
```

You can also assign a user name and password to the proxy, if necessary. Please consult the documentation for `com.jcraft.jsch.Proxy` to discover all options.

Setting preferred SFTP authentication method

If you want to explicitly specify the list of authentication methods that should be used by sftp component, use `preferredAuthentications` option. If for example you would like Camel to attempt to authenticate with private/public SSH key and fallback to user/password authentication in the case when no public key is available, use the following route configuration: `from("sftp://localhost:9999/root?username=admin&password=admin&preferredAuthentications=publickey,password").to("bean:processFile");`

Consuming a single file using a fixed name

When you want to download a single file and know the file name, you can use `fileName=myFileName.txt` to tell Camel the name of the file to download. By default the consumer will still do a FTP LIST command to do a directory listing and then filter these files based on the `fileName` option. Though in this use-case it may be desirable to turn off the directory listing by setting `useList=false`. For example the user account used to login to the FTP server may not have permission to do a FTP LIST command. So you can turn off this with `useList=false`, and then provide the fixed name of the file to download with `fileName=myFileName.txt`, then the FTP consumer can still download the file. If the file for some reason does not exist, then Camel will by default throw an exception, you can turn this off and ignore this by setting `ignoreFileNotFoundException=true`.

For example to have a Camel route that picks up a single file, and deletes it after use you can write

```
from("ftp://admin@localhost:21/nolist?&
password=admin&stepwise=false&useList=false&ignoreFileNotFoundException=true&fileName=report.txt&delete=true")
.to("activemq:queue:report");
```

Notice that we have used all the options we talked above.

You can also use this with `ConsumerTemplate`. For example to download a single file (if it exists) and grab the file content as a String type:

```
String data = template.retrieveBodyNoWait("ftp://admin@localhost:21/nolist?&
password=admin&stepwise=false&useList=false&ignoreFileNotFoundException=true&fileName=report.txt&delete=true",
String.class);
```

Debug logging

This component has log level **TRACE** that can be helpful if you have problems.

Endpoint See Also

- File2

CAMEL COMPONENTS FOR GOOGLE APP ENGINE

This component is deprecated and will be removed from Camel 2.18 onwards.

Tutorials

- A good starting point for using Camel on GAE is the Tutorial for Camel on Google App Engine
- The OAuth tutorial demonstrates how to implement OAuth in web applications.

The Camel components for Google App Engine (GAE) are part of the camel-gae project and provide connectivity to GAE's cloud computing services. They make the GAE cloud computing environment accessible to applications via Camel interfaces. Following this pattern for other cloud computing environments could make it easier to port Camel applications from one cloud computing provider to another. The following table lists the cloud computing services provided by Google and the supporting Camel components. The documentation of each component can be found by following the link in the *Camel Component* column.

GAE service	Camel component	Component description
URL fetch service	ghttp	Provides connectivity to the GAE URL fetch service but can also be used to receive messages from servlets.
Task queuing service	gtask	Supports asynchronous message processing on GAE by using the task queuing service as message queue.
Mail service	gmail	Supports sending of emails via the GAE mail service. Receiving mails is not supported yet but will be added later.
Memcache service		Not supported yet.
Xmpp service		Not supported yet.
Images service		Not supported yet.
Datastore service		Not supported yet.
Accounts service	gauth glogin	These components interact with the Google Accounts API for authentication and authorization. Google Accounts is not specific to Google App Engine but is often used by GAE applications for implementing security. The gauth component is used by web applications to implement a Google-specific OAuth consumer. This component can also be used to OAuth-enable non-GAE web applications. The glogin component is used by Java clients (outside GAE) for programmatic login to GAE applications. For instructions how to protect GAE applications against unauthorized access refer to the Security for Camel on GAE application page.

Camel context

Setting up a SpringCamelContext on Google App Engine differs between Camel 2.1 and higher versions. The problem is that usage of the Camel-specific Spring configuration XML schema from the `http://camel.apache.org/schema/spring` namespace requires JAXB and Camel 2.1 depends on a Google App Engine SDK version that doesn't support JAXB yet. This limitation has been removed since Camel 2.2.

JMX must be disabled in any case because the `javax.management` package isn't on the App Engine JRE whitelist.

Camel 2.1

camel-gae 2.1 comes with the following CamelContext implementations.

- `org.apache.camel.component.gae.context.GaeDefaultCamelContext` (extends `org.apache.camel.impl.DefaultCamelContext`)
- `org.apache.camel.component.gae.context.GaeSpringCamelContext` (extends `org.apache.camel.spring.SpringCamelContext`)

Both disable JMX before startup. The `GaeSpringCamelContext` additionally provides setter methods adding route builders as shown in the next example.

```
Listing 1. appctx.xml

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="camelContext"
          class="org.apache.camel.component.gae.context.GaeSpringCamelContext">
        <property name="routeBuilder" ref="myRouteBuilder" />
    </bean>

    <bean id="myRouteBuilder"
          class="org.example.MyRouteBuilder">
    </bean>

</beans>
```

Alternatively, use the `routeBuilders` property of the `GaeSpringCamelContext` for setting a list of route builders. Using this approach, a `SpringCamelContext` can be configured on GAE without the need for JAXB.

Camel 2.2 or higher

With Camel 2.2 or higher, applications can use the `http://camel.apache.org/schema/spring` namespace for configuring a `SpringCamelContext` but still need to disable JMX. Here's an example.

Listing 1. appctx.xml

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

http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

    <camel:camelContext id="camelContext">
        <camel:jmxAgent id="agent" disabled="true" />
        <camel:routeBuilder ref="myRouteBuilder"/>
    </camel:camelContext>

    <bean id="myRouteBuilder"
          class="org.example.MyRouteBuilder">
    </bean>
</beans>
```

The web.xml

Running Camel on GAE requires usage of the `CamelHttpTransportServlet` from `camel-servlet`. The following example shows how to configure this servlet together with a Spring application context XML file.

Listing 1. web.xml

```
<web-app
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation=
  "http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">

    <servlet>
        <servlet-name>CamelServlet</servlet-name>
        <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>appctx.xml</param-value>
        </init-param>
    </servlet>

    <!--
        Mapping used for external requests
    -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/camel/*</url-pattern>
    </servlet-mapping>

    <!--
        Mapping used for web hooks accessed by task queueing service.
    -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/worker/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The location of the Spring application context XML file is given by the `contextConfigLocation` init parameter. The `appctx.xml` file must be on the classpath. The servlet mapping makes the Camel application accessible under `http://<appname>.appspot.com/camel/...` when deployed to Google App Engine where `<appname>` must be replaced by a real GAE application name. The second servlet mapping is used internally by the task queueing service for background processing via web hooks. This mapping is relevant for the gtask component and is explained there in more detail.

HAZELCAST COMPONENT

Available as of Camel 2.7

The `hazelcast` component allows you to work with the Hazelcast distributed data grid / cache. Hazelcast is a in memory data grid, entirely written in Java (single jar). It offers a great palette of different data stores like map, multi map (same key, n values), queue, list and atomic number. The main reason to use Hazelcast is its simple cluster support. If you have enabled multicast on your network you can run a cluster with hundred nodes with no extra configuration. Hazelcast can simply configured to add additional features like n copies between nodes (default is 1), cache persistence, network configuration (if needed), near cache, eviction and so on. For more information consult the Hazelcast documentation on <http://www.hazelcast.com/docs.jsp>.

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

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

```
</dependency>
```

URI format

```
hazelcast:[ map | multimap | queue | topic | seda | set | atomicvalue | instance | list | ringbuffer]:cachename[?options]
```

Topic support is available as of Camel 2.15.

RingBuffer support is available as of Camel 2.16.

Options

Name	Required	Description
hazelcastInstance	No	Camel 2.14: The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.
hazelcastInstanceName	No	Camel 2.16: The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.
operation	-1	To specify a default operation to use, if no operation header has been provided. deprecated use defaultOperation instead.
defaultOperation	-1	Camel 2.15: To specify a default operation to use, if no operation header has been provided.

You have to use the second prefix to define which type of data store you want to use.

Sections

1. Usage of #map
2. Usage of #multimap
3. Usage of #queue
4. Usage of #topic
5. Usage of #list
6. Usage of #seda
7. Usage of atomic number
8. Usage of #cluster support (instance)
9. Usage of #replicatedmap
10. Usage of #ringbuffer

Usage of Map

map cache producer - to("hazelcast:map:foo")

If you want to store a value in a map you can use the map cache producer. The map cache producer provides 5 operations (put, get, update, delete, query). For the first 4 you have to provide the operation inside the "hazelcast.operation.type" header variable. In Java DSL you can use the constants from org.apache.camel.component.hazelcast.HazelcastConstants.

Header Variables for the request message:

Name	Type	Description
hazelcast.operation.type	String	valid values are: put, delete, get, update, query
hazelcast.objectId	String	the object id to store / find your object inside the cache (not needed for the query operation)

Header variables have changed in Camel 2.8

Name	Type	Description
CamelHazelcastOperationType	String	valid values are: put, delete, get, update, query Version 2.8 From Camel 2.16: getAll, putIfAbsent, clear.
CamelHazelcastobjectId	String	the object id to store / find your object inside the cache (not needed for the query operation) Version 2.8

You can call the samples with:

```
template.sendBodyAndHeader("direct:[put|get|update|delete|query]", "my-foo", HazelcastConstants.OBJECT_ID, "4711");
```

Sample for put:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PUT_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
  <from uri="direct:put" />
  <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
  <setHeader headerName="hazelcast.operation.type">
    <constant>put</constant>
  </setHeader>
  <to uri="hazelcast:map:foo" />
</route>
```

Sample for get:**Java DSL:**

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
  <from uri="direct:get" />
  <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
  <setHeader headerName="hazelcast.operation.type">
    <constant>get</constant>
  </setHeader>
  <to uri="hazelcast:map:foo" />
  <to uri="seda:out" />
</route>
```

Sample for update:**Java DSL:**

```
from("direct:update")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.UPDATE_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
  <from uri="direct:update" />
  <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
  <setHeader headerName="hazelcast.operation.type">
    <constant>update</constant>
  </setHeader>
  <to uri="hazelcast:map:foo" />
</route>
```

Sample for delete:**Java DSL:**

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.DELETE_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
  <from uri="direct:delete" />
  <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
  <setHeader headerName="hazelcast.operation.type">
    <constant>delete</constant>
  </setHeader>
  <to uri="hazelcast:map:foo" />
</route>
```

Sample for query**Java DSL:**

```
from("direct:query")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.QUERY_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
    <from uri="direct:query" />
    <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>query</constant>
    </setHeader>
    <to uri="hazelcast:map:foo" />
    <to uri="sedat:out" />
</route>
```

For the query operation Hazelcast offers a SQL like syntax to query your distributed map.

```
String q1 = "bar > 1000";
template.sendBodyAndHeader("direct:query", null, HazelcastConstants.QUERY, q1);
```

map cache consumer - from("hazelcast:map:foo")

Hazelcast provides event listeners on their data grid. If you want to be notified if a cache will be manipulated, you can use the map consumer. There're 4 events: **put**, **update**, **delete** and **envict**. The event type will be stored in the "hazelcast.listener.action" header variable. The map consumer provides some additional information inside these variables:

Header Variables inside the response message:

Name	Type	Description
hazelcast.listener.time	Long	time of the event in millis
hazelcast.listener.type	String	the map consumer sets here "cachelistener"
hazelcast.listener.action	String	type of event - here added , updated , envict and removed
hazelcast.objectid	String	the oid of the object
hazelcast.cache.name	String	the name of the cache - e.g. "foo"
hazelcast.cache.type	String	the type of the cache - here map

Header variables have changed in Camel 2.8

Name	Type	Description
CamelHazelcastListenerTime	Long	time of the event in millis Version 2.8
CamelHazelcastListenerType	String	the map consumer sets here "cachelistener" Version 2.8
CamelHazelcastListenerAction	String	type of event - here added , updated , envict and removed . Version 2.8
CamelHazelcastObjectId	String	the oid of the object Version 2.8
CamelHazelcastCacheName	String	the name of the cache - e.g. "foo" Version 2.8
CamelHazelcastCacheType	String	the type of the cache - here map Version 2.8

The object value will be stored within **put** and **update** actions inside the message body.

Here's a sample:

```
fromF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.log("object...")
.choice()
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
        .log("...added")
        .to("mock:added")
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ENVICTED))
        .log("...envict")
        .to("mock:envict")
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.UPDATED))
        .log("...updated")
        .to("mock:updated")
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
        .log("...removed")
        .to("mock:removed")
    .otherwise()
        .log("fail!");
```

Usage of Multi Map**multimap cache producer - to("hazelcast:multimap:foo")**

A multimap is a cache where you can store n values to one key. The multimap producer provides 4 operations (put, get, removevalue, delete).

Header Variables for the request message:

Name	Type	Description
------	------	-------------

`hazelcast.operation.type` String valid values are: put, get, removevalue, delete
`hazelcast.objectId` String the object id to store / find your object inside the cache

Header variables have changed in Camel 2.8

Name	Type	Description
<code>CamelHazelcastOperationType</code>	String	valid values are: put, get, removevalue, delete Version 2.8 From Camel 2.16: clear.
<code>CamelHazelcastObjectId</code>	String	the object id to store / find your object inside the cache Version 2.8

Sample for put:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PUT_OPERATION))
.to(String.format("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX));
```

Spring DSL:

```
<route>
    <from uri="direct:put" />
    <log message="put..."/>
    <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>put</constant>
    </setHeader>
    <to uri="hazelcast:multimap:foo" />
</route>
```

Sample for removevalue:

Java DSL:

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.REMOVEVALUE_OPERATION))
.to("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:removevalue" />
    <log message="removevalue..."/>
    <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>removevalue</constant>
    </setHeader>
    <to uri="hazelcast:multimap:foo" />
</route>
```

To remove a value you have to provide the value you want to remove inside the message body. If you have a multimap object {key: "4711" values: { "my-foo", "my-bar"}} you have to put "my-foo" inside the message body to remove the "my-foo" value.

Sample for get:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.GET_OPERATION))
.to("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.to("sed:out");
```

Spring DSL:

```
<route>
    <from uri="direct:get" />
    <log message="get..."/>
    <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>get</constant>
    </setHeader>
    <to uri="hazelcast:multimap:foo" />
    <to uri="sed:out" />
</route>
```

Sample for delete:

Java DSL:

```
from("direct:delete")
```

```
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.DELETE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:delete" />
    <log message="delete..."/>
    <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>delete</constant>
    </setHeader>
    <to uri="hazelcast:multimap:foo" />
</route>
```

you can call them in your test class with:

```
template.sendBodyAndHeader("direct:[put|get|removevalue|delete]", "my-foo", HazelcastConstants.OBJECT_ID, "4711");
```

multimap cache consumer - from("hazelcast:multimap:foo")

For the multimap cache this component provides the same listeners / variables as for the map cache consumer (except the update and eviction listener). The only difference is the **multimap** prefix inside the URI. Here is a sample:

```
fromF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.log("object...")
.choice()
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
        .log("...added")
        .to("mock:added")
    //,when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.EVICTED))
    //,log("...evicted")
    //,to("mock:evicted")
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
        .log("...removed")
        .to("mock:removed")
    .otherwise()
        .log("fail!");
```

Header Variables inside the response message:

Name	Type	Description
hazelcast.listener.time	Long	time of the event in millis
hazelcast.listener.type	String	the map consumer sets here "cachelistener"
hazelcast.listener.action	String	type of event - here added and removed (and soon evicted)
hazelcast.objectId	String	the oid of the object
hazelcast.cache.name	String	the name of the cache - e.g. "foo"
hazelcast.cache.type	String	the type of the cache - here multimap

Eviction will be added as feature, soon (this is a Hazelcast issue).

Header variables have changed in Camel 2.8

Name	Type	Description
CamelHazelcastListenerTime	Long	time of the event in millis Version 2.8
CamelHazelcastListenerType	String	the map consumer sets here "cachelistener" Version 2.8
CamelHazelcastListenerAction	String	type of event - here added and removed (and soon evicted) Version 2.8
CamelHazelcastObjectId	String	the oid of the object Version 2.8
CamelHazelcastCacheName	String	the name of the cache - e.g. "foo" Version 2.8
CamelHazelcastCacheType	String	the type of the cache - here multimap Version 2.8

Usage of Queue**Queue producer – to("hazelcast:queue:foo")**

The queue producer provides 6 operations (add, put, poll, peek, offer, removevalue).

Sample for add:

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.ADD_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Sample for put:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PUT_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Sample for poll:

```
from("direct:poll")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.POLL_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Sample for peek:

```
from("direct:peek")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PEEK_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Sample for offer:

```
from("direct:offer")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.OFFER_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Sample for removevalue:

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.REMOVEVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Queue consumer – from("hazelcast:queue:foo")

The queue consumer provides 2 operations (add, remove).

```
fromF("hazelcast:%smm", HazelcastConstants.QUEUE_PREFIX)
.log("object...")
.choice()
.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
.log("...added")
.to("mock:added")
.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
.log("...removed")
.to("mock:removed")
.otherwise()
.log("fail!");
```

Usage of Topic**Topic producer – to("hazelcast:topic:foo")**

The topic producer provides only one operation (publish).

Sample for publish:

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PUBLISH_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.PUBLISH_OPERATION);
```

Topic consumer – from("hazelcast:topic:foo")

The topic consumer provides only one operation (received). This component is supposed to support multiple consumption as it's expected when it comes to topics so you are free to have as much consumers as you need on the same hazelcast topic.

```
fromF("hazelcast:%sfoo", HazelcastConstants.TOPIC_PREFIX)
.choice()
.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.RECEIVED))
.log("...message received")
.otherwise()
.log("...this should never have happened")
```

Usage of List

List producer – to("hazelcast:list:foo")

The list producer provides 4 operations (add, addAll, set, get, removevalue, removeAll, clear).

Sample for add:

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.ADD_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

Sample for get:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX)
.to("seda:out");
```

Sample for setvalue:

```
from("direct:set")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.SETVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

Sample for removevalue:

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.REMOVEVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

Note that **CamelHazelcastObjectIndex** header is used for indexing purpose.

The list consumer provides 2 operations (add, remove).List consumer – from("hazelcast:list:foo")

```
fromF("hazelcast:%smm", HazelcastConstants.LIST_PREFIX)
.log("object...")
.choice()
.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
.log("...added")
.to("mock:added")
.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
.log("...removed")
.to("mock:removed")
.otherwise()
.log("fail!");
```

Usage of SEDA

SEDA component differs from the rest components provided. It implements a work-queue in order to support asynchronous SEDA architectures, similar to the core "SEDA" component.

SEDA producer – to("hazelcast:seda:foo")

The SEDA producer provides no operations. You only send data to the specified queue.

Name	default value	Description
transferExchange	false	Camel 2.8.0: if set to true the whole Exchange will be transferred. If header or body contains not serializable objects, they will be skipped.

Java DSL :

```
from("direct:foo")
.to("hazelcast:seda:foo");
```

Spring DSL :

```
<route>
    <from uri="direct:start" />
    <to uri="hazelcast:seda:foo" />
</route>
```

SEDA consumer – from("hazelcast:seda:foo")

The SEDA consumer provides no operations. You only retrieve data from the specified queue.

Name	default value	Description
pollInterval	1000	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown. (deprecated from Camel 2.15 onwards, use pollTimeout instead).
pollTimeout	1000	Camel 2.15: The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.
concurrentConsumers	1	To use concurrent consumers polling from the SEDA queue.
transferExchange	false	Camel 2.8.0: if set to true the whole Exchange will be transferred. If header or body contains not serializable objects, they will be skipped.
transacted	false	Camel 2.10.4: if set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.

Java DSL :

```
from("hazelcast:seda:foo")
.to("mock:result");
```

Spring DSL:

```
<route>
  <from uri="hazelcast:seda:foo" />
  <to uri="mock:result" />
</route>
```

Usage of Atomic Number

There is no consumer for this endpoint!

atomic number producer - to("hazelcast:atomicnumber:foo")

An atomic number is an object that simply provides a grid wide number (long). The operations for this producer are setvalue (set the number with a given value), get, increase (+1), decrease (-1) and destroy.

Header Variables for the request message:

Name	Type	Description
hazelcast.operation.type	String	valid values are: setvalue, get, increase, decrease, destroy

Header variables have changed in Camel 2.8

Name	Type	Description
CamelHazelcastOperationType	String	valid values are: setvalue, get, increase, decrease, destroy Available as of Camel version 2.8

Sample for set:

Java DSL:

```
from("direct:set")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.SETVALUE_OPERATION))
.to("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
  <from uri="direct:set" />
  <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
  <setHeader headerName="hazelcast.operation.type">
    <constant>setvalue</constant>
  </setHeader>
  <to uri="hazelcast:atomicvalue:foo" />
</route>
```

Provide the value to set inside the message body (here the value is 10): `template.sendBody("direct:set", 10);`

Sample for get:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.GET_OPERATION))
.to("hazelcast:%$foo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
  <from uri="direct:get" />
  <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
  <setHeader headerName="hazelcast.operation.type">
    <constant>get</constant>
  </setHeader>
  <to uri="hazelcast:atomicvalue:foo" />
</route>
```

You can get the number with long body = template.requestBody("direct:get", null, Long.class);.

Sample for increment:**Java DSL:**

```
from("direct:increment")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.INCREMENT_OPERATION))
.to("hazelcast:%$foo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
  <from uri="direct:increment" />
  <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
  <setHeader headerName="hazelcast.operation.type">
    <constant>increment</constant>
  </setHeader>
  <to uri="hazelcast:atomicvalue:foo" />
</route>
```

The actual value (after increment) will be provided inside the message body.

Sample for decrement:**Java DSL:**

```
from("direct:decrement")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.DECREMENT_OPERATION))
.to("hazelcast:%$foo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
  <from uri="direct:decrement" />
  <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
  <setHeader headerName="hazelcast.operation.type">
    <constant>decrement</constant>
  </setHeader>
  <to uri="hazelcast:atomicvalue:foo" />
</route>
```

The actual value (after decrement) will be provided inside the message body.

Sample for destroy

There's a bug inside Hazelcast. So this feature may not work properly. Will be fixed in 1.9.3.

Java DSL:

```
from("direct:destroy")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.DESTROY_OPERATION))
.to("hazelcast:%$foo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
  <from uri="direct:destroy" />
  <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
  <setHeader headerName="hazelcast.operation.type">
    <constant>destroy</constant>
  </setHeader>
  <to uri="hazelcast:atomicvalue:foo" />
</route>
```

cluster support

This endpoint provides no producer!

instance consumer - from("hazelcast:instance:foo")

Hazelcast makes sense in one single "server node", but it's extremely powerful in a clustered environment. The instance consumer fires if a new cache instance will join or leave the cluster.

Here's a sample:

```
fromF("hazelcast:$foo", HazelcastConstants.INSTANCE_PREFIX)
.log("instance...")
.choice()
.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
.log("...added")
.to("mock:added")
.otherwise()
.log("...removed")
.to("mock:removed");
```

Each event provides the following information inside the message header:

Header Variables inside the response message:

Name	Type	Description
hazelcast.listener.time	Long	time of the event in millis
hazelcast.listener.type	String	the map consumer sets here "instancelistener"
hazelcast.listener.action	String	type of event - here added or removed
hazelcast.instance.host	String	host name of the instance
hazelcast.instance.port	Integer	port number of the instance

Header variables have changed in Camel 2.8

Name	Type	Description
CamelHazelcastListenerTime	Long	time of the event in millis Version 2.8
CamelHazelcastListenerType	String	the map consumer sets here "instancelistener" Version 2.8
CamelHazelcastListenerActionn	String	type of event - here added or removed . Version 2.8
CamelHazelcastInstanceHost	String	host name of the instance Version 2.8
CamelHazelcastInstancePort	Integer	port number of the instance Version 2.8

Using hazelcast reference

By its name

```
<bean id="hazelcastLifecycle" class="com.hazelcast.core.LifecycleService"
factory-bean="hazelcastInstance" factory-method="getLifecycleService"
destroy-method="shutdown" />

<bean id="config" class="com.hazelcast.config.Config">
<constructor>arg type="java.lang.String" value="HZ.INSTANCE" />
</bean>

<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast" factory-method="newHazelcastInstance">
<constructor>arg type="com.hazelcast.config.Config" ref="config"/>
</bean>
<camelContext xmlns="http://camel.apache.org/schema/spring">
<route id="testHazelcastInstanceBeanRefPut">
<from uri="direct:testHazelcastInstanceBeanRefPut"/>
<setHeader headerName="CamelHazelcastOperationType">
<constant>put</constant>
</setHeader>
<to uri="hazelcast:map:testmap?hazelcastInstanceName=HZ.INSTANCE"/>
</route>
<route id="testHazelcastInstanceBeanRefGet">
<from uri="direct:testHazelcastInstanceBeanRefGet" />
<setHeader headerName="CamelHazelcastOperationType">
<constant>get</constant>
</setHeader>
<to uri="hazelcast:map:testmap?hazelcastInstanceName=HZ.INSTANCE"/>
<to uri="seda:out" />
</route>
</camelContext>
```

By instance

```
<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast"
factory-method="newHazelcastInstance" />
<bean id="hazelcastLifecycle" class="com.hazelcast.core.LifecycleService"
factory-bean="hazelcastInstance" factory-method="getLifecycleService"
destroy-method="shutdown" />
```

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route id="testHazelcastInstanceBeanRefPut">
    <from uri="direct:testHazelcastInstanceBeanRefPut"/>
    <setHeader headerName="CamelHazelcastOperationType">
      <constant>put</constant>
    </setHeader>
    <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
  </route>
  <route id="testHazelcastInstanceBeanRefGet">
    <from uri="direct:testHazelcastInstanceBeanRefGet" />
    <setHeader headerName="CamelHazelcastOperationType">
      <constant>get</constant>
    </setHeader>
    <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
    <to uri="seda:out" />
  </route>
</camelContext>
```

Publishing hazelcast instance as an OSGI service

If operating in an OSGI container and you would want to use one instance of hazelcast across all bundles in the same container. You can publish the instance as an OSGI service and bundles using the cache al need is to reference the service in the hazelcast endpoint.

Bundle A create an instance and publishes it as an OSGI service

```
<bean id="config" class="com.hazelcast.config.FileSystemXmlConfig">
  <argument type="java.lang.String" value="${hazelcast.config}"/>
</bean>

<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast" factory-method="newHazelcastInstance">
  <argument type="com.hazelcast.config.Config" ref="config"/>
</bean>

<!-- publishing the hazelcastInstance as a service -->
<service ref="hazelcastInstance" interface="com.hazelcast.core.HazelcastInstance" />
```

Bundle B uses the instance

```
<!-- referencing the hazelcastInstance as a service -->
<reference ref="hazelcastInstance" interface="com.hazelcast.core.HazelcastInstance" />

<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  <route id="testHazelcastInstanceBeanRefPut">
    <from uri="direct:testHazelcastInstanceBeanRefPut"/>
    <setHeader headerName="CamelHazelcastOperationType">
      <constant>put</constant>
    </setHeader>
    <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
  </route>
  <route id="testHazelcastInstanceBeanRefGet">
    <from uri="direct:testHazelcastInstanceBeanRefGet" />
    <setHeader headerName="CamelHazelcastOperationType">
      <constant>get</constant>
    </setHeader>
    <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
    <to uri="seda:out" />
  </route>
</camelContext>
```

Usage of Replicated map

Available from Camel 2.16

replicatedmap cache producer - to("hazelcast:replicatedmap:foo")

A replicated map is a weakly consistent, distributed key-value data structure with no data partition. The replicatedmap producer provides 4 operations (put, get, delete, clear).

Header Variables for the request message:

Name	Type	Description
hazelcast.operation.type	String	valid values are: put, get, delete, clear
hazelcast.objectId	String	the object id to store / find your object inside the cache

Header variables have changed in Camel 2.8

Name	Type	Description
CamelHazelcastOperationType	String	valid values are: put, get, removevalue, delete Version 2.8

CamelHazelcastobjectId String the object id to store / find your object inside the cache **Version 2.8**

Sample for put:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PUT_OPERATION))
.to(String.format("hazelcast:%sbar", HazelcastConstants.REPLICATEDMAP_PREFIX));
```

Spring DSL:

```
<route>
<from uri="direct:put" />
<log message="put..."/>
<!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
<setHeader headerName="hazelcast.operation.type">
<constant>put</constant>
</setHeader>
<to uri="hazelcast:replicatedmap:foo" />
</route>
```

Sample for get:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.GET_OPERATION))
.to("hazelcast:%sbar", HazelcastConstants.REPLICATEDMAP_PREFIX)
.to("sedao:out");
```

Spring DSL:

```
<route>
<from uri="direct:get" />
<log message="get..."/>
<!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
<setHeader headerName="hazelcast.operation.type">
<constant>get</constant>
</setHeader>
<to uri="hazelcast:replicatedmap:foo" />
<to uri="sedao:out" />
</route>
```

Sample for delete:

Java DSL:

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.DELETE_OPERATION))
.to("hazelcast:%sbar", HazelcastConstants.REPLICATEDMAP_PREFIX);
```

Spring DSL:

```
<route>
<from uri="direct:delete" />
<log message="delete..."/>
<!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
<setHeader headerName="hazelcast.operation.type">
<constant>delete</constant>
</setHeader>
<to uri="hazelcast:replicatedmap:foo" />
</route>
```

you can call them in your test class with:

```
template.sendBodyAndHeader("direct:[put|get|delete|clear]", "my-foo", HazelcastConstants.OBJECT_ID, "4711");
```

replicatedmap cache consumer - from("hazelcast:replicatedmap:foo")

For the multimap cache this component provides the same listeners / variables as for the map cache consumer (except the update and eviction listener). The only difference is the **multimap** prefix inside the URI. Here is a sample:

```
fromF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.log("object...")
.choice()
.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
.log("...added")
.to("mock:added")
//.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.EVICTED))
//.log("...evicted")
//.to("mock:evicted")
.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
.log("...removed")
```

```

        .to("mock:removed")
.otherwise()
.log("fail!");

```

Header Variables inside the response message:

Name	Type	Description
hazelcast.listener.time	Long	time of the event in millis
hazelcast.listener.type	String	the map consumer sets here "cachelistener"
hazelcast.listener.action	String	type of event - here added and removed (and soon evicted)
hazelcast.objectId	String	the oid of the object
hazelcast.cache.name	String	the name of the cache - e.g. "foo"
hazelcast.cache.type	String	the type of the cache - here replicatedmap

Eviction will be added as feature, soon (this is a Hazelcast issue).

Header variables have changed in Camel 2.8

Name	Type	Description
CamelHazelcastListenerTime	Long	time of the event in millis Version 2.8
CamelHazelcastListenerType	String	the map consumer sets here "cachelistener" Version 2.8
CamelHazelcastListenerAction	String	type of event - here added and removed (and soon evicted) Version 2.8
CamelHazelcastobjectId	String	the oid of the object Version 2.8
CamelHazelcastCacheName	String	the name of the cache - e.g. "foo" Version 2.8
CamelHazelcastCacheType	String	the type of the cache - here replicatedmap Version 2.8

Usage of Ringbuffer

Available from Camel 2.16

ringbuffer cache producer - to("hazelcast:ringbuffer:foo")

Ringbuffer is a distributed data structure where the data is stored in a ring-like structure. You can think of it as a circular array with a certain capacity. The ringbuffer producer provides 5 operations (add, readonceHead, readonceTail, remainingCapacity, capacity).

Header Variables for the request message:

Name	Type	Description
hazelcast.operation.type	String	valid values are: add, readonceHead, readonceTail, remainingCapacity, capacity
Header variables have changed in Camel 2.8		
Name	Type	Description
CamelHazelcastOperationType	String	valid values are: put, get, removevalue, delete Version 2.8
CamelHazelcastobjectId	String	the object id to store / find your object inside the cache Version 2.8

Sample for put:

Java DSL:

```

from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.ADD_OPERATION))
.to(String.format("hazelcast:%sbar", HazelcastConstants.RINGBUFFER_PREFIX));

```

Spring DSL:

```

<route>
  <from uri="direct:put" />
  <log message="put.."/>
  <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
  <setHeader headerName="hazelcast.operation.type">
    <constant>add</constant>
  </setHeader>
  <to uri="hazelcast:ringbuffer:foo" />
</route>

```

Sample for readonce from head:

Java DSL:

```

from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.READ_ONCE_HEAD_OPERATION))

```

```
.toF("hazelcast:%sbar", HazelcastConstants.RINGBUFFER_PREFIX)
.to("seda:out");
```

HDFS COMPONENT

Available as of Camel 2.8

The **hdfs** component enables you to read and write messages from/to an HDFS file system. HDFS is the distributed file system at the heart of Hadoop.

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

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

URI format

```
hdfs://hostname[:port]/[path][?options]
```

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

The path is treated in the following way:

1. as a consumer, if it's a file, it just reads the file, otherwise if it represents a directory it scans all the file under the path satisfying the configured pattern. All the files under that directory must be of the same type.
2. as a producer, if at least one split strategy is defined, the path is considered a directory and under that directory the producer creates a different file per split named using the configured UuidGenerator.

When consuming from hdfs then in normal mode, a file is split into chunks, producing a message per chunk. You can configure the size of the chunk using the chunkSize option. If you want to read from hdfs and write to a regular file using the file component, then you can use the fileMode=Append to append each of the chunks together.

Options

Name	Default Value	Description
overwrite	true	The file can be overwritten.
append	false	Append to existing file. Notice that not all HDFS file systems support the append option.
buffersize	4096	The buffer size used by HDFS.
replication	3	The HDFS replication factor.
blocksize	67108864	The size of the HDFS blocks.
filetype	NORMAL_FILE	It can be SEQUENCE_FILE, MAP_FILE, ARRAY_FILE or BLOOMMAP_FILE, see Hadoop.
filesystemtype	HDFS	It can be LOCAL for local filesystem.
keytype	NULL	The type for the key in case of sequence or map files. See below.
valuetype	TEXT	The type for the key in case of sequence or map files. See below.
splitstrategy		A string describing the strategy on how to split the file based on different criteria. See below.
openedsuffix	opened	When reading a file, it is opened with this suffix. It is useful to read it during the writing phase.
readclosedsuffix	closed	Once the file has been read, it is renamed with this suffix. It is useful to read it during the writing phase.
initialdelay	0	For the consumer, how much to wait (milliseconds) before to start scanning the directory.
delay	0	The interval (milliseconds) between the directory scans.
pattern	*	The pattern used for scanning the directory.
chunksize	4096	When reading a normal file, this is split into chunks producing a message per chunk.
connectonstartup	true	Camel 2.9.3/2.10.1: Whether to connect to the HDFS file system on starting the producer/consumer. If false then the connection is created on-demand. Notice that HDFS may take up till 15 minutes to establish a connection, as it has hardcoded 45 x 20 sec redelivery. By setting this option to false allows your application to startup, and not block for up till 15 minutes.
owner		Camel 2.13/2.12.4: The file owner must match this owner for the consumer to pickup the file. Otherwise the file is skipped.

KeyType and ValueType

- NULL it means that the key or the value is absent
- BYTE for writing a byte, the java Byte class is mapped into a BYTE
- BYTES for writing a sequence of bytes. It maps the java ByteBuffer class
- INT for writing java integer
- FLOAT for writing java float
- LONG for writing java long
- DOUBLE for writing java double
- TEXT for writing java strings

BYTES is also used with everything else, for example, in Camel a file is sent around as an InputStream, int this case is written in a sequence file or a map file as a sequence of bytes.

Splitting Strategy

In the current version of Hadoop opening a file in append mode is disabled since it's not very reliable. So, for the moment, it's only possible to create new files. The Camel HDFS endpoint tries to solve this problem in this way:

- If the split strategy option has been defined, the hdfs path will be used as a directory and files will be created using the configured UuidGenerator
- Every time a splitting condition is met, a new file is created.
The splitStrategy option is defined as a string with the following syntax:
`splitStrategy=<ST>:<value>,<ST>:<value>,*`

where <ST> can be:

- BYTES a new file is created, and the old is closed when the number of written bytes is more than <value>
- MESSAGES a new file is created, and the old is closed when the number of written messages is more than <value>
- IDLE a new file is created, and the old is closed when no writing happened in the last <value> milliseconds

note that this strategy currently requires either setting an IDLE value or setting the HdfsConstants.HDFS_CLOSE header to false to use the BYTES/MESSAGES configuration...otherwise, the file will be closed with each message
for example:

```
hdfs://localhost/tmp/simple-file?splitStrategy=IDLE:1000,BYTES:5
```

it means: a new file is created either when it has been idle for more than 1 second or if more than 5 bytes have been written. So, running hadoop fs -ls /tmp/simple-file you'll see that multiple files have been created.

Message Headers

The following headers are supported by this component:

Producer only

Header	Description
<small>Camel 2.13: Specifies the name of the file to write (relative to the endpoint path). The name can be a string or an Expression object. Only relevant when not using a split strategy.</small>	

Controlling to close file stream

Available as of Camel 2.10.4

When using the HDFS producer **without** a split strategy, then the file output stream is by default closed after the write. However you may want to keep the stream open, and only explicitly close the stream later. For that you can use the header `HdfsConstants.HDFS_CLOSE` (value = "CamelHdfsClose") to control this. Setting this value to a boolean allows you to explicit control whether the stream should be closed or not.

Notice this does not apply if you use a split strategy, as there are various strategies that can control when the stream is closed.

Using this component in OSGI

This component is fully functional in an OSGI environment, however, it requires some actions from the user. Hadoop uses the thread context class loader in order to load resources. Usually, the thread context classloader will be the bundle class loader of the bundle that contains the routes. So, the default configuration files need to be visible from the bundle class loader. A typical way to deal with it is to keep a copy of core-default.xml in your bundle root. That file can be found in the hadoop-common.jar.

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.

Note that Camel also ships with a JPA component. The JPA component abstracts from the underlying persistence provider and allows you to work with Hibernate, OpenJPA or EclipseLink.

Sending to the endpoint

Sending POJOs to the hibernate endpoint inserts entities into the database. The body of the message is assumed to be an entity bean that you have mapped to a relational table using the hibernate .hbm.xml files.

If the body does not contain an entity bean, use a Message Translator in front of the endpoint to perform the necessary conversion first.

Consuming from the endpoint

Consuming messages removes (or updates) entities in the database. This allows you to use a database table as a logical queue; consumers take messages from the queue and then delete/update them to logically remove them from the queue.

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

If you would rather perform some update on the entity to mark it as processed (such as to exclude it from a future query) then you can annotate a method with `@Consumed` which will be invoked on your entity bean when the entity bean is consumed.

URI format

```
hibernate:[entityClassName][?options]
```

For sending to the endpoint, the `entityClassName` is optional. If specified it is used to help use the type conversion to ensure the body is of the correct type.

For consuming the `entityClassName` is mandatory.

You can append query options to the URI in the following format, `?option=value&option2=value&...`

Options

Name	Default Value	Description
<code>entityType</code>	<code>entityClassName</code>	Is the provided <code>entityClassName</code> from the URI.
<code>consumeDelete</code>	<code>true</code>	Option for <code>HibernateConsumer</code> only. Specifies whether or not the entity is deleted after it is consumed.
<code>consumeLockPriority</code>	<code>true</code>	Option for <code>HibernateConsumer</code> only. Specifies whether or not to use exclusive locking of each entry while processing the results from the pooling.
<code>flushMode</code>	<code>none</code>	Option for <code>HibernateConsumer</code> . Flush mode for the session.
<code>maxResults</code>	<code>1</code>	Option for <code>HibernateConsumer</code> only. Set the maximum number of results to retrieve on the Query.
<code>consumer.delay</code>	<code>500</code>	Option for <code>HibernateConsumer</code> only. Delay in millis between each poll.
<code>consumer.initialDelay</code>	<code>1000</code>	Option for <code>HibernateConsumer</code> only. Millis before polling starts.
<code>consumer.userFixedDelay</code>	<code>false</code>	Option for <code>HibernateConsumer</code> only. Set to <code>true</code> to use fixed delay between polls, otherwise fixed rate is used. See <code>ScheduledExecutorService</code> in <code>JDK</code> for details.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Hibernate Example

HIPCHAT COMPONENT

Available as of Camel 2.15.0

The Hipchat component supports producing and consuming messages from/to Hipchat service.

Prerequisites

You must have a valid Hipchat user account and get a personal access token that you can use to produce/consume messages.

URI Format

```
hipchat://[:host][:port]?options
```

You can append query options to the URI in the following format, `?options=value&option2=value&...`

URI Options

Name	Default Value	Context	Required	Producer/Consumer	Description
<code>protocol</code>	<code>null</code>	Shared	Yes	Both	Default protocol to connect to the Hipchat server such as <code>http</code> or <code>https</code>
<code>host</code>	<code>null</code>	Shared	Yes	Both	The API host of the Hipchat to connect to, such as <code>api.hipchat.com</code>
<code>port</code>	<code>80</code>	Shared	No	Both	The port to connect to on the Hipchat host
<code>authToken</code>	<code>null</code>	Shared	Yes	Both	Authorization token(personal access token) obtained from Hipchat
<code>delay</code>	<code>5000</code>	Shared	No	Consumer	The poll interval in millisec for

					consuming messages from consumeUsers provided. Please read about rate limits before decreasing this.
consumeUsers	null	Shared	No	Consumer	Comma separated list of user @Mentions or emails whose messages to the owner of authToken must be consumed

Scheduled Poll Consumer

This component implements the ScheduledPollConsumer. Only the last message from the provided 'consumeUsers' are retrieved and sent as Exchange body. If you do not want the same message to be retrieved again when there are no new messages on next poll then you can add the idempotent consumer as shown below. All the options on the ScheduledPollConsumer can also be used for more control on the consumer.

```
@Override
public void configure() throws Exception {
    String hipchatEndpointUri = "hipchat://?authToken=XXXX&consumeUsers=@Joe,@John";
    from(hipchatEndpointUri)
        .idempotentConsumer(
            simple("${in.header.HipchatMessageDate} ${in.header.HipchatFromUser}",
                  MemoryIdempotentRepository.memoryIdempotentRepository(200))
        )
        .to("mock:result");
}
```

Message headers set by the Hipchat consumer

Header	Constant	Type	Description
HipchatFromUser	HipchatConstants.FROM_USER	String	The body has the message that was sent from this user to the owner of authToken
HipchatMessageDate	HipchatConstants.MESSAGE_DATE	String	The date message was sent. The format is ISO-8601 as present in the Hipchat response.
HipchatFromUserResponseStatus	HipchatConstants.FROM_USER_RESPONSE_STATUS	StatusLine	The status of the API response received

Hipchat Producer

Producer can send messages to both Room's and User's simultaneously. The body of the exchange is sent as message. Sample usage is shown below. Appropriate headers needs to be set.

```
@Override
public void configure() throws Exception {
    String hipchatEndpointUri = "hipchat://?authToken=XXXX";
    from("direct:start")
        .to(hipchatEndpointUri)
        .to("mock:result");
}
```

Message headers evaluated by the Hipchat producer

Header	Constant	Type	Description
HipchatToUser	HipchatConstants.TO_USER	String	The Hipchat user to which the message needs to be sent
HipchatToRoom	HipchatConstants.TO_ROOM	String	The Hipchat room to which the message needs to be sent
HipchatMessageFormat	HipchatConstants.MESSAGE_FORMAT	String	Valid formats are 'text' or 'html'. Default: 'text'
HipchatMessageBackgroundColor	HipchatConstants.MESSAGE_BACKGROUND_COLOR	String	Valid color values are 'yellow', 'green', 'red', 'purple', 'gray', 'random'. Default: 'yellow' (Room Only)
HipchatTriggerNotification	HipchatConstants.TRIGGER_NOTIFY	String	Valid values are 'true' or 'false'. Whether this message should trigger a user notification (change the tab color, play a sound, notify mobile phones, etc). Default: 'false' (Room Only)

Message headers set by the Hipchat producer

Header	Constant	Type	Description
HipchatToUserResponseStatus	HipchatConstants.TO_USER_RESPONSE_STATUS	StatusLine	The status of the API response received when message sent to the user
HipchatFromUserResponseStatus	HipchatConstants.TO_ROOM_RESPONSE_STATUS	StatusLine	The status of the API response received when message sent to the room

Dependencies

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

```
Listing 1. pom.xml

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

where \${camel-version} must be replaced by the actual version of Camel (2.15.0 or higher)

HL7 COMPONENT

The HL7 component is used for working with the HL7 MLLP protocol and HL7 v2 messages using the HAPI library.

This component supports the following:

- HL7 MLLP codec for Mina
- HL7 MLLP codec for Netty4 from **Camel 2.15** onwards
- Type Converter from/to HAPI and String
- HL7 DataFormat using the HAPI library
- Even more ease-of-use as it's integrated well with the camel-mina2 component.

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

```
xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-hl7</artifactId> <version>x.x.x</version> <!-- use the same version as your Camel core version --> </dependency>
```

HL7 MLLP protocol

HL7 is often used with the HL7 MLLP protocol, which is a text based TCP socket based protocol. This component ships with a Mina and Netty4 Codec that conforms to the MLLP protocol so you can easily expose an HL7 listener accepting HL7 requests over the TCP transport layer. To expose a HL7 listener service, the camel-mina2 or camel-netty4 component is used with the HL7MLLPCodec (mina2) or HL7MLLPNettyDecoder/HL7MLLPNettyEncoder (Netty4).

HL7 MLLP codec can be configured as follows:

confluenceTableSmall

Name	Default Value	Description
startByte	0x0b	The start byte spanning the HL7 payload.
endByte1	0x1c	The first end byte spanning the HL7 payload.
endByte2	0x0d	The 2nd end byte spanning the HL7 payload.
charset	JVM Default	The encoding (a charset name) to use for the codec. If not provided, Camel will use the JVM default Charset.
produceString	true	(as of Camel 2.14.1) If true, the codec creates a string using the defined charset. If false, the codec sends a plain byte array into the route, so that the HL7 Data Format can determine the actual charset from the HL7 message content.
convertLFtoCR	false	Will convert \n to \r (0x0d, 13 decimal) as HL7 stipulates \r as segment terminators. The HAPI library requires the use of \r.

Exposing an HL7 listener using Mina

In the Spring XML file, we configure a mina2 endpoint to listen for HL7 requests using TCP on port 8888:

```
xml<endpoint id="hl7MinaListener" uri="mina2:tcp://localhost:8888?sync=true&codec=#hl7codec"/>
```

sync=true indicates that this listener is synchronous and therefore will return a HL7 response to the caller. The HL7 codec is setup with **codec=#hl7codec**. Note that hl7codec is just a Spring bean ID, so it could be named mygreatcodecforhl7 or whatever. The codec is also set up in the Spring XML file:

```
xml<bean id="hl7codec" class="org.apache.camel.component.hl7.HL7MLLPCodec"> <property name="charset" value="iso-8859-1"/></bean>
```

The endpoint **hl7MinaListener** can then be used in a route as a consumer, as this Java DSL example illustrates:

```
java from("hl7MinaListener").beanRef("patientLookupService");
```

This is a very simple route that will listen for HL7 and route it to a service named **patientLookupService**. This is also Spring bean ID, configured in the Spring XML as:

```
xml<bean id="patientLookupService" class="com.mycompany.healthcare.service.PatientLookupService"/>
```

The business logic can be implemented in POJO classes that do not depend on Camel, as shown here:

```
javainport ca.uhn.hl7v2.HL7Exception; import ca.uhn.hl7v2.model.Message; import ca.uhn.hl7v2.model.v24.segment.QRD;
public class PatientLookupService { public Message lookupPatient(Message input) throws HL7Exception { QRD qrd =
(QRD)input.get("QRD"); String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue(); // find patient data based on the patient id and create a HL7 model object with the response Message response = ... create and set response data return response
} }
```

Exposing an HL7 listener using Netty (available from Camel 2.15 onwards)

In the Spring XML file, we configure a netty4 endpoint to listen for HL7 requests using TCP on port 8888:

```
xml<endpoint id="hl7NettyListener" uri="netty4:tcp://localhost:8888?sync=true&encoder=#hl7encoder&decoder=#hl7decoder"/>
```

sync=true indicates that this listener is synchronous and therefore will return a HL7 response to the caller. The HL7 codec is setup with **encoder=#hl7encoder** and **decoder=#hl7decoder**. Note that hl7encoder and hl7decoder are just bean IDs, so they could be named differently. The beans can be set in the Spring XML file:

```
xml <bean id="hl7decoder" class="org.apache.camel.component.hl7.HL7MLPNettyDecoderFactory"/> <bean id="hl7encoder" class="org.apache.camel.component.hl7.HL7MLPNettyEncoderFactory"/>
The endpoint hl7NettyListener can then be used in a route as a consumer, as this Java DSL example illustrates:
java from("hl7NettyListener").beanRef("patientLookupService");
```

HL7 Model using java.lang.String or byte[]

The HL7 MLP codec uses plain String as its data format. Camel uses its Type Converter to convert to/from strings to the HAPI HL7 model objects, but you can use the plain String objects if you prefer, for instance if you wish to parse the data yourself.

As of Camel 2.14.1 you can also let both the Mina and Netty codecs use a plain byte[] as its data format by setting the produceString property to false. The Type Converter is also capable of converting the byte[] to/from HAPI HL7 model objects.

HL7v2 Model using HAPI

The HL7v2 model uses Java objects from the HAPI library. Using this library, you can encode and decode from the EDI format (ER7) that is mostly used with HL7v2.

The sample below is a request to lookup a patient with the patient ID 0101701234.

```
MSH|^~\&|MYSENDER|MYRECEIVER|MYAPPLICATION||200612211200||QRY^A19|1234|P|2.4
QRD|200612211200|R||GetPatient||1^RD|0101701234|DEM|
Using the HL7 model you can work with a ca.uhn.hl7v2.model.Message object, e.g. to retrieve a patient ID:
javaMessage msg = exchange.getIn().getBody(Message.class); QRD qrd = (QRD)msg.get("QRD"); String patientId =
qrd.getWhoSubjectFilter(0).getIDNumber().getValue(); // 0101701234
This is powerful when combined with the HL7 listener, because you don't have to work with byte[], String or any other simple object formats. You can just use the HAPI HL7v2 model objects. If you know the message type in advance, you can be more type-safe:
javaQRY_A19 msg = exchange.getIn().getBody(QRY_A19.class); String patientId =
msg.getQRD().getWhoSubjectFilter(0).getIDNumber().getValue();
```

HL7 DataFormat

Message Headers

The unmarshal operation adds these fields from the MSH segment as headers on the Camel message:

confluenceTableSmall

Key	MSH field	Example
CamelHL7SendingApplication	MSH-3	MYSERVER
CamelHL7SendingFacility	MSH-4	MYSERVERAPP
CamelHL7ReceivingApplication	MSH-5	MYCLIENT
CamelHL7ReceivingFacility	MSH-6	MYCLIENTAPP
CamelHL7Timestamp	MSH-7	20071231235900
CamelHL7Security	MSH-8	null
CamelHL7MessageType	MSH-9-1	ADT
CamelHL7TriggerEvent	MSH-9-2	A01
CamelHL7MessageControl	MSH-10	1234
CamelHL7ProcessingId	MSH-11	P
CamelHL7VersionId	MSH-12	2.4
CamelHL7Context	-	(Camel 2.14) contains the HapiContext that was used to parse the message
CamelHL7Charset	MSH-18	(Camel 2.14) UNICODE UTF-8

All headers except CamelHL7Context are String types. If a header value is missing, its value is null.

Options

The HL7 Data Format supports the following options:

confluenceTableSmall

Option	Default	Description
validate	true	Whether the HAPI Parser should validate the message using the default validation rules. It is recommended to use the parser or hapiContext option and initialize it with the desired HAPI ValidationContext
parser	ca.uhn.hl7v2.parser.GenericParser	Custom parser to be used. Must be of type

ca.uhn.hl7v2.parser.Parser. Note that GenericParser also allows to parse XML-encoded HL7v2 messages

hapicontext ca.uhn.hl7v2.DefaultHapiContext

Camel 2.14: Custom HAPI context that can define a custom parser, custom ValidationContext etc. This gives you full control over the HL7 parsing and rendering process.

Dependencies

To use HL7 in your Camel routes you'll need to add a dependency on **camel-hl7** listed above, which implements this data format.

The HAPI library is split into a base library and several structure libraries, one for each HL7v2 message version:

- v2.1 structures library
- v2.2 structures library
- v2.3 structures library
- v2.3.1 structures library
- v2.4 structures library
- v2.5 structures library
- v2.5.1 structures library
- v2.6 structures library

By default **camel-hl7** only references the HAPI base library. Applications are responsible for including structure libraries themselves. For example, if an application works with HL7v2 message versions 2.4 and 2.5 then the following dependencies must be added:

```
xml<dependency> <groupId>ca.uhn.hapi</groupId> <artifactId>hapi-structures-v24</artifactId> <version>2.2</version> <!-- use the same version as your hapi-base version --> </dependency> <dependency> <groupId>ca.uhn.hapi</groupId> <artifactId>hapi-structures-v25</artifactId> <version>2.2</version> <!-- use the same version as your hapi-base version --> </dependency>
```

Alternatively, an OSGi bundle containing the base library, all structures libraries and required dependencies (on the bundle classpath) can be downloaded from the central Maven repository.

```
xml<dependency> <groupId>ca.uhn.hapi</groupId> <artifactId>hapi-osgi-base</artifactId> <version>2.2</version> </dependency>
```

Terser language

HAPI provides a Terser class that provides access to fields using a commonly used terse location specification syntax. The Terser language allows to use this syntax to extract values from messages and to use them as expressions and predicates for filtering, content-based routing etc.

Sample:

```
javaimport static org.apache.camel.component.hl7.HL7.terser; ... // extract patient ID from field QRD-8 in the QRY_A19 message above and put into message header from("direct:test1") .setHeader("PATIENT_ID",terser("QRD-8(0)-1")) .to("mock:test1"); // continue processing if extracted field equals a message header from("direct:test2") .filter(terser("QRD-8(0)-1").isEqualTo(header("PATIENT_ID")) .to("mock:test2");
```

HL7 Validation predicate

Often it is preferable to first parse a HL7v2 message and in a separate step validate it against a HAPI ValidationContext.

Sample:

```
javaimport static org.apache.camel.component.hl7.HL7.messageConformsTo; import ca.uhn.hl7v2.validation.impl.DefaultValidation; ... // Use standard or define your own validation rules ValidationContext defaultContext = new DefaultValidation(); // Throws PredicateValidationException if message does not validate from("direct:test1") .validate(messageConformsTo(defaultContext)) .to("mock:test1");
```

HL7 Validation predicate using the HapiContext (Camel 2.14)

The HAPI Context is always configured with a ValidationContext (or a ValidationRuleBuilder), so you can access the validation rules indirectly. Furthermore, when unmarshalling the HL7DataFormat forwards the configured HAPI context in the

CamelHL7Context header, and the validation rules of this context can be easily reused:

```
javaimport static org.apache.camel.component.hl7.HL7.messageConformsTo; import static org.apache.camel.component.hl7.HL7.messageConforms ... HapiContext hapiContext = new DefaultHapiContext(); hapiContext.getParserConfiguration().setValidating(false); // don't validate during parsing // customize HapiContext some more ... e.g. enforce that PID-8 in ADT_A01 messages of version 2.4 is not empty ValidationRuleBuilder builder = new ValidationRuleBuilder() { @Override protected void configure() { forVersion(Version.V24).message("ADT", "A01").terser("PID-8", not(empty())); } }; hapiContext.setValidationRuleBuilder(builder); HL7DataFormat hl7 = new HL7DataFormat(); hl7.setHapiContext(hapiContext); from("direct:test1") .unmarshal(hl7) // uses the GenericParser returned from the HapiContext .validate(messageConforms()) // uses the validation rules returned from the HapiContext // equivalent with .validate(messageConformsTo(hapiContext)) // route continues from here
```

HL7 Acknowledgement expression

A common task in HL7v2 processing is to generate an acknowledgement message as response to an incoming HL7v2 message, e.g. based on a validation result. The ack expression lets us accomplish this very elegantly:

```
javaimport static org.apache.camel.component.hl7.HL7.messageConformsTo; import static
org.apache.camel.component.hl7.HL7.ack; import ca.uhn.hl7v2.validation.impl.DefaultValidation; ... // Use standard or define your
own validation rules ValidationContext defaultContext = new DefaultValidation(); from("direct:test1")
.onException(Exception.class) .handled(true) .transform(ack()) // auto-generates negative ack because of exception in Exchange
.end() .validate(messageConformsTo(defaultContext)) // do something meaningful here ... // acknowledgement .transform(ack())
```

More Samples

In the following example, a plain string HL7 request is sent to an HL7 listener that sends back a response:

```
{snippet:id=e2|lang=java|url=camel/trunk/components/camel-
hl7/src/test/java/org/apache/camel/component/hl7/HL7MLLPCodecTest.java}
```

In the next sample, HL7 requests from the HL7 listener are routed to the business logic, which is implemented as plain POJO registered in the registry as h17service.

```
{snippet:id=e2|lang=java|url=camel/trunk/components/camel-
hl7/src/test/java/org/apache/camel/component/hl7/HL7RouteTest.java}
```

Then the Camel routes using the RouteBuilder may look as follows:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-
hl7/src/test/java/org/apache/camel/component/hl7/HL7RouteTest.java}
```

Note that by using the HL7 DataFormat the Camel message headers are populated with the fields from the MSH segment. The headers are particularly useful for filtering or content-based routing as shown in the example above.

Endpoint See Also

HTTP COMPONENT

The **http:** component provides HTTP based endpoints for consuming external HTTP resources (as a client to call external servers using HTTP).

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

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

URI Format

http:hostname[:port][/resourceUri][?param1=value1][¶m2=value2]

Will by default use port 80 for HTTP and 443 for HTTPS.

camel-http vs camel-jetty

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

Examples

Call the URL with the body using POST and return response as the OUT message. If body is null call URL using GET and return response as OUT message:

Java DSL	Spring DSL
from("direct:start").to("http://myhost/mypath");	xml<from uri="direct:start"/> <to uri="http://oldhost"/>

You can override the HTTP endpoint URI by adding a header. Camel will call the http://newhost. This is very handy for e.g. REST URLs:

Java DSL
javafrom("direct:start") .setHeader(Exchange.HTTP_URI, simple("http://myserver/orders/\${header.orderId}")) .to("http://dummyhost");

URI parameters can either be set directly on the endpoint URI or as a header:

Java DSL
javafrom("direct:start") .to("http://oldhost?order=123&detail=short"); from("direct:start") .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short")) .to("http://oldhost");

Set the HTTP request method to POST:

Java DSL	Spring DSL
from("direct:start") .setHeader(Exchange.HTTP_METHOD,	xml<from uri="direct:start"/> <setHeader headerName="CamelHttpMethod"> <constant>POST</constant> </setHeader> <to

```
constant("POST"))
.to("http://www.google.com");
uri="http://www.google.com"/> <to uri="mock:results"/>
```

HttpEndpoint Options

confluenceTableSmall

Name	Default Value	Description
throwExceptionOnFailure	true	Option to disable throwing the <code>HttpoperationFailedException</code> in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.
bridgeEndpoint	false	If the option is true, <code>HttpProducer</code> will ignore the <code>Exchange.HTTP_URI</code> header, and use the endpoint's URI for request. You may also set <code>throwExceptionOnFailure=false</code> to ensure all responses are propagated back to the <code>HttpProducer</code> . From Camel 2.3 : when true the <code>HttpProducer</code> and <code>CamelServlet</code> will skip gzip processing when <code>content-encoding=gzip</code> .
disableStreamCache	false	When false the <code>DefaultHttpBinding</code> will copy the request input stream into a stream cache and put it into message body which allows it to be read more than once. When true the <code>DefaultHttpBinding</code> will set the request input stream direct into the message body. From Camel 2.17 : this options is now also support by the producer to allow using the response stream directly instead of stream caching as by default.
<code>httpbindingref</code>	null	Deprecated and removed in Camel 2.17: Reference to a <code>org.apache.camel.component.http.HttpBinding</code> in the Registry. Use the <code>httpBinding</code> option instead.
<code>httpbinding</code>	null	From Camel 2.3 : reference to a <code>org.apache.camel.component.http.HttpBinding</code> in the Registry.
<code>httpclientconfigurerref</code>	null	Deprecated and removed in Camel 2.17: Reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry. Use the <code>httpClientConfigurer</code> option instead.
<code>httpClientConfigurer</code>	null	From Camel 2.3 : reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry.
<code>httpClient.XXX</code>	null	Use this to option to configure the underlying <code>HttpClientParams</code> . Example: <code>httpClient.soTimeout=5000</code> will set the <code>so_TIMEOUT</code> to 5 seconds.
<code>clientConnectionManager</code>	null	To use a custom <code>org.apache.http.conn.ClientConnectionManager</code> .
<code>transferException</code>	false	From Camel 2.6 : If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a <code>application/x-java-serialized-object</code> content type (for example using Jetty or SERVLET Camel components). On the producer side the exception will be deserialized and thrown as is, instead of the <code>HttpoperationFailedException</code> . The caused exception will be serialized.
<code>headerFilterstrategy</code>	null	From Camel 2.11 : reference to a instance of <code>org.apache.camel.spi.HeaderFilterStrategy</code> in the Registry. It will be used to apply the custom <code>headerFilterStrategy</code> on the new create <code>HttpEndpoint</code> .
<code>urlRewrite</code>	null	From Camel 2.11: Producer only! Refers to a custom <code>org.apache.camel.component.http.UrlRewrite</code> which allows you to rewrite URLs when you bridge/proxy endpoints. See more details at UrlRewrite and How to use Camel as a HTTP proxy between a client and server .
<code>eagerCheckContentAvailable</code>	false	From Camel 2.15.3/2.16: Consumer only! Whether to eager check whether the HTTP requests has content when <code>content-length=0</code> or is not present. This option should be set to <code>true</code> for those HTTP clients that do not send streamed data.
<code>copyHeaders</code>	true	From Camel 2.16 : if this option is true then <code>IN</code> exchange headers will be copied to <code>OUT</code> exchange headers according to copy strategy. Setting this to <code>false</code> , allows to only include the headers from the HTTP response (not propagating <code>IN</code> headers).

okstatusCodeRange	200-299	From Camel 2.16 : the range of HTTP status codes for which a response is considered a success. The values are inclusive. The range must be in the form from-to, dash included.
ignoreResponseBody	false	From Camel 2.16 : when true the <code>HttpProducer</code> will not read the response body nor cache the input stream.
cookieHandler	null	From Camel 2.19 : configure a cookie handler to maintain a HTTP session

Authentication and Proxy

The following authentication options can also be set on the `HttpEndpoint`:

Name	Default Value	Description
authmethod	null	Authentication method, either as <code>Basic</code> , <code>Digest</code> or <code>NTLM</code> .
authMethodPriority	null	Priority of authentication methods. Is a list separated with comma. For example: <code>Basic,Digest</code> to exclude NTLM.
authusername	null	Username for authentication.
authpassword	null	Password for authentication.
authdomain	null	Domain for NTLM authentication.
authHost	null	Optional host for NTLM authentication.
proxyHost	null	The proxy host name.
proxyPort	null	The proxy port number.
proxyAuthMethod	null	Authentication method for proxy, either as <code>Basic</code> , <code>Digest</code> or <code>NTLM</code> .
proxyAuthUsername	null	Username for proxy authentication.
proxyAuthPassword	null	Password for proxy authentication.
proxyAuthDomain	null	Domain for proxy NTLM authentication.
proxyAuthHost	null	Optional host for proxy NTLM authentication.

When using authentication you **must** provide the choice of method for the `authMethod` or `authProxyMethod` options. You can configure the proxy and authentication details on either the `HttpComponent` or the `HttpEndpoint`. Values provided on the `HttpEndpoint` will take precedence over `HttpComponent`. Its most likely best to configure this on the `HttpComponent` which allows you to do this once.

The HTTP component uses convention over configuration which means that if you have not explicit set a `authMethodPriority` then it will fallback and use the select(ed) `authMethod` as priority as well. So if you use `authMethod.Basic` then the `authMethodPriority` will be `Basic` only.

Note: `camel-http` is based on `HttpClient v3.x` and as such has only limited support for what is known as `NTLMv1`, the early version of the NTLM protocol. It does not support `NTLMv2` at all. `camel-http4` has support for `NTLMv2`.

HttpComponent Options

confluenceTableSmall

Name	Default Value	Description
httpBinding	null	To use a custom <code>org.apache.camel.component.http.HttpBinding</code> .
httpClientConfigurer	null	To use a custom <code>org.apache.camel.component.http.HttpClientConfigurer</code> .
httpConnectionManager	null	To use a custom <code>org.apache.commons.httpclient.HttpConnectionManager</code> .
httpConfiguration	null	To use a custom <code>org.apache.camel.component.http.HttpConfiguration</code> .
allowJavaSerializedObject	false	Camel 2.16.1/2.15.5: Whether to allow java serialization when a request uses <code>context-type=application/x-java-serialized-object</code> . If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.

`HttpConfiguration` contains all the options listed in the table above under the section *HttpConfiguration - Setting Authentication and Proxy*.

Message Headers

confluenceTableSmall

Name	Type	Description
<code>Exchange.HTTP_URI</code>	String	URI to call. Will override existing URI set directly on the

		endpoint. This URI is the URI of the HTTP server to call. Its not the same as the Camel endpoint URI, where you can configure endpoint options such as security etc. This header does not support that, its only the URI of the HTTP server.
Exchange.HTTP_METHOD	String	HTTP method/verb to use. Can be one of: <ul style="list-style-type: none">• GET• POST• PUT• DELETE• HEAD• OPTIONS• TRACE
Exchange.HTTP_PATH	String	The request URI's path. The header will be used to build the request URI with the <code>HTTP_URI</code> . From Camel 2.3.0 : if the path starts with a /, the <code>HttpProducer</code> will try to find the relative path based on the <code>Exchange.HTTP_BASE_URI</code> header or the <code>exchange.getFromEndpoint().getEndpointUri()</code> ;
Exchange.HTTP_QUERY	String	URI parameters. Will override existing URI parameters set directly on the endpoint.
Exchange.HTTP_RESPONSE_CODE	int	The HTTP response code from the external server. Is 200 for OK.
Exchange.HTTP_CHARACTER_ENCODING	String	Character encoding.
Exchange.CONTENT_TYPE	String	The HTTP content type. Is set on both the IN and OUT message to provide a content type, such as <code>text/html</code> .
Exchange.CONTENT_ENCODING	String	The HTTP content encoding. Is set on both the IN and OUT message to provide a content encoding, such as <code>gzip</code> .
Exchange.HTTP_SERVLET_REQUEST	<code>HttpServletRequest</code>	The <code>HttpServletRequest</code> object.
Exchange.HTTP_SERVLET_RESPONSE	<code>HttpServletResponse</code>	The <code>HttpServletResponse</code> object.
Exchange.HTTP_PROTOCOL_VERSION	String	From Camel 2.5 : You can set the HTTP protocol version with this header, e.g., <code>HTTP/1.0</code> . If the header is not present the <code>HttpProducer</code> will use the default value <code>HTTP/1.1</code> .

Note: The header names above are constants. For the spring DSL you have to use the value of the constant instead of the name.

Message Body

Camel will store the HTTP response from the external server on the OUT body. All headers from the IN message will be copied to the OUT message, so headers are preserved during routing. Additionally Camel will add the HTTP response headers as well to the OUT message headers.

Response Code

Camel will handle according to the HTTP response code:

- Response code is in the range 100..299, Camel regards it as a success response.
- Response code is in the range 300..399, Camel regards it as a redirection response and will throw a `HttpOperationFailedException` with the information.
- Response code is 400+, Camel regards it as an external server failure and will throw a `HttpoperationFailedException` with the information.

`throwExceptionOnFailure`

The option, `throwExceptionOnFailure`, can be set to `false` to prevent the `HttpoperationFailedException` from being thrown for failed response codes. This allows you to get any response from the remote server.

There is a sample below demonstrating this.

`HttpOperationFailedException`

This exception contains the following information:

- The HTTP status code.
- The HTTP status line (text of the status code).

- Redirect location, if server returned a redirect.
- Response body as a `java.lang.String`, if server provided a body as response.

Calling Using GET or POST

The following algorithm is used to determine if either GET or POST HTTP method should be used:

1. Use method provided in header.
2. GET if query string is provided in header.
3. GET if endpoint is configured with a query string.
4. POST if there is data to send (body is not null).
5. GET otherwise.

How To Access The HttpServletRequest and HttpServletResponse

You can get access to these two using the Camel type converter system using:

```
javaHttpServletRequest request = exchange.getIn().getBody(HttpServletRequest.class); HttpServletRequest response =
exchange.getIn().getBody(HttpServletResponse.class);
```

Using Client Timeout - SO_TIMEOUT

See the unit test in this link

MORE EXAMPLES

Configuring a Proxy

Java DSL

```
from("direct:start").to("http://oldhost?proxyHost=www.myproxy.com&proxyPort=80");
```

There is also support for proxy authentication via the `proxyUsername` and `proxyPassword` options.

Using Proxy Settings Outside of the URI

Java DSL	Spring DSL
<pre>context.getProperties().put("http.proxyHost", "172.168.18.9"); context.getProperties().put("http.proxyPort" "8080");</pre>	<pre><camelContext> <properties> <property key="http.proxyHost" value="172.168.18.9"/> <property key="http.proxyPort" value="8080"/> </properties> </camelContext></pre>

Options on Endpoint will override options on the context.

Configuring charset

If you are using POST to send data you can configure the charset:
`.setProperty(Exchange.CHARSET_NAME, "iso-8859-1");`

Sample with Scheduled Poll

The sample polls the Google homepage every 10 seconds and write the page to the file `message.html`:

```
javafrom("timer://foo?fixedRate=true&delay=0&period=10000").to("http://www.google.com")
.setHeader(FileComponent.HEADER_FILE_NAME, "message.html").to("file:target/google");
```

Getting the Response Code

You can get the HTTP response code from the HTTP component by getting the value from the `out` message header with `Exchange.HTTP_RESPONSE_CODE`:

```
javaExchange exchange = template.send("http://www.google.com/search", new Processor() { public void process(Exchange exchange) throws Exception { exchange.getIn().setHeader(Exchange.HTTP_QUERY, constant("hl=en&q=activemq")); } });
Message out = exchange.getOut(); int responseCode = out.getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);
```

Using throwExceptionOnFailure=false To Obtain All Server Responses

In the route below we want to route a message that we enrich with data returned from a remote HTTP call. As we want all responses from the remote server, we set the `throwExceptionOnFailure=false` so we get any response in the AggregationStrategy. As the code is based on a unit test that simulates a HTTP status code 404, there is some assertion code

```
etc.{snippet:id=e1|lang=java|url=camel/tags/camel-2.2.0/components/camel-jetty/src/test/java/org/apache/camel/component/jetty/JettySimplifiedHandle404Test.java}
```

Disabling Cookies

To disable cookies you can set the HTTP Client to ignore cookies by adding this URI option:
`httpClient.cookiePolicy=ignoreCookies`

Advanced Usage

If you need more control over the HTTP producer you should use the `HttpComponent` where you can set various classes to give you custom behavior.

Setting MaxConnectionsPerHost

The HTTP Component has a `org.apache.commons.httpclient.HttpConnectionManager` where you can configure various global configuration for the given component. By global, we mean that any endpoint the component creates has the same shared `HttpConnectionManager`. So, if we want to set a different value for the max connection per host, we need to define it on the `HttpComponent` and *not* on the endpoint URI that we usually use. So here comes:

First, we define the `http` component in Spring XML. Yes, we use the same scheme name, `http`, because otherwise Camel will auto-discover and create the component with default settings. What we need is to overrule this so we can set our options. In the sample below we set the max connection to 5 instead of the default of 2.{snippet:id=e1|lang=xml|url=camel/tags/camel-2.2.0/tests/camel-itest/src/test/resources/org/apache/camel/itest/http/HttpMaxConnectionPerHostTest-context.xml}And then we can just use it as we normally do in our routes:{snippet:id=e2|lang=xml|url=camel/tags/camel-2.2.0/tests/camel-itest/src/test/resources/org/apache/camel/itest/http/HttpMaxConnectionPerHostTest-context.xml}

Using Pre-Emptive Authentication

If an HTTP server should fail to respond correctly with an expected 401 Authorization Required response for a failed authentication attempt a client can instead use preemptive authentication by specifying the URI option:
`httpClient.authenticationPreemptive=true`.

Accepting Self-Signed Certificates From Remote Server

See this link from a mailing list discussion with some code to outline how to do this with the Apache Commons HTTP API.

Setting up SSL for HTTP Client

Using the JSSE Configuration Utility

From **Camel 2.8**: the `HTTP4` component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the `HTTP4` component.

The version of the Apache HTTP client used in this component resolves SSL/TLS information from a global "protocol" registry. This component provides an implementation,

`org.apache.camel.component.http.SSLContextParameters$secureProtocolSocketFactory`, of the HTTP client's protocol socket factory in order to support the use of the Camel JSSE Configuration utility. The following example demonstrates how to configure the protocol registry and use the registered protocol information in a route.

```
javaKeyStoreParameters ksp = new KeyStoreParameters(); ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword"); KeyManagersParameters kmp = new KeyManagersParameters(); kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword"); SSLContextParameters scp = new SSLContextParameters(); scp.setKeyManagers(kmp);
ProtocolSocketFactory factory = new SSLContextParameters$SecureProtocolSocketFactory(scp);
Protocol.registerProtocol("https", new Protocol("https", factory, 443)); from("direct:start") .to("https://mail.google.com/mail/")
.to("mock:results");
```

Configuring Apache HTTP Client Directly

Basically `camel-http` component is built on the top of Apache HTTP client, and you can implement a custom `org.apache.camel.component.http.HttpClientConfigurer` to do some configuration on the HTTP client if you need full control of it.

However, if you *just* want to specify the keystore and truststore you can do this with Apache HTTP `HttpClientConfigurer`, for example:

```
javaProtocol authhttps = new Protocol("https", new AuthSSLProtocolSocketFactory(new URL("file:my.keystore"), "mypassword",
new URL("file:my.truststore"), "mypassword"), 443); Protocol.registerProtocol("https", authhttps);
```

And then you need to create a class that implements `HttpClientConfigurer`, and registers HTTPS protocol providing a keystore or truststore per example above. Then, from your Camel RouteBuilder class you can hook it up like so:

```
javaHtppComponent httpComponent = getContext().getComponent("http", HttpComponent.class);
httpComponent.setHttpClientConfigurer(new MyHttpClientConfigurer());
```

If you are doing this using the Spring DSL, you can specify your `HttpClientConfigurer` using the URI. For example:

```
xml-bean id="myHttpClientConfigurer" class="my.https.HttpClientConfigurer"/> <to uri="https://myhostname.com:443/myURL?httpClientConfigurerRef=myHttpClientConfigurer"/>
```

As long as you implement the `HttpClientConfigurer` and configure your keystore and truststore as described above, it will work fine.

Endpoint See Also

- Jetty

iBATIS

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

The Apache iBatis project is no longer active. The project is moved outside Apache and is now known as the MyBatis project. Therefore we encourage users to use MyBatis instead. This camel-ibatis component will be removed in Camel 3.0.

iBatis do not support Spring 4.x. So you can only use Spring 3.x or older with iBatis.

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

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

URI format

`ibatis:statementName[?options]`

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

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

This component will by default load the iBatis SqlMapConfig file from the root of the classpath and expected named as `SqlMapConfig.xml`.

It uses Spring resource loading so you can define it using `classpath`, `file` or `http` as prefix to load resources with those schemes.

In Camel 2.2 you can configure this on the `iBatisComponent` with the `setSqlMapConfig(String)` method.

Options

confluenceTableSmall

Option	Type	Default	Description
<code>consumer.onConsume</code>	String	null	Statements to run after consuming. Can be used, for example, to update rows as they have been consumed and processed in Camel. See sample later. Multiple statements can be separated with commas.
<code>consumer.useIterator</code>	boolean	true	If true each row returned when polling will be processed individually. If false the entire list of data is set as the IN body.
<code>consumer.routeEmptyResultSet</code>	boolean	false	Sets whether empty result set should be routed or not. By default, empty result sets are not routed.
<code>statementType</code>	StatementType	null	Mandatory to specify for IbatisProducer control which iBatis <code>SqlMapClient</code> method to invoke. The enum values are <code>QueryForObject</code> , <code>QueryForList</code> , <code>Insert</code> , <code>Update</code> , <code>Delete</code> .
<code>maxMessagesPerPoll</code>	int	0	An integer to define a maximum messages to gather per poll. By default no maximum is set. Can be used to set limit of e.g. 1000 to avoid when starting the server that there are thousands of files. Set a value of 0 or negative to disable it.
<code>isolation</code>	String	TRANSACTION_REPEATABLE_READ	Camel 2.9: A String that defines the transaction isolation level of the will be used. Allowed values are <code>TRANSACTION_NONE</code> , <code>TRANSACTION_READ_UNCOMMITTED</code> , <code>TRANSACTION_READ_COMMITTED</code>

TRANSACTION_REPEATABLE_READ
TRANSACTION_SERIALIZABLE

isolation string TRANSACTION_REPEATABLE_READ	Camel 2.9: A String that defines the transaction isolation level of the will be used. Allowed values are TRANSACTION_NONE, TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_READ, TRANSACTION_SERIALIZABLE
--	---

Message Headers

Camel will populate the result message, either IN or OUT with a header with the operationName used:
confluenceTableSmall

Header	Type	Description
CamelIBatisStatementName	String	The statementName used (for example: insertAccount).
CamelIBatisResult	object	The response returned from iBatis in any of the operations. For instance an INSERT could return the auto-generated key, or number of rows etc.

Message Body

The response from iBatis will only be set as body if it's a **SELECT** statement. That means, for example, for **INSERT** statements Camel will not replace the body. This allows you to continue routing and keep the original body. The response from iBatis is always stored in the header with the key **CamelIBatisResult**.

Samples

For example if you wish to consume beans from a JMS queue and insert them into a database you could do the following:
from("activemq:queue:newAccount").to("ibatis:insertAccount?statementType=Insert");

Notice we have to specify the **statementType**, as we need to instruct Camel which **sqlMapClient** operation to invoke.

Where **insertAccount** is the iBatis ID in the SQL map file:
xml <!-- Insert example, using the Account parameter class --> <insert id="insertAccount" parameterClass="Account"> insert into ACCOUNT(ACC_ID, ACC_FIRST_NAME, ACC_LAST_NAME, ACC_EMAIL) values (#id#, #firstName#, #lastName#, #emailAddress#) </insert>

Using StatementType for better control of iBatis

When routing to an iBatis endpoint you want more fine grained control so you can control whether the SQL statement to be executed is a **SELECT**, **UPDATE**, **DELETE** or **INSERT** etc. So for instance if we want to route to an iBatis endpoint in which the IN body contains parameters to a **SELECT** statement we can do:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-ibatis/src/test/java/org/apache/camel/component/ibatis/IbatisQueryForObjectTest.java}
```

In the code above we can invoke the iBatis statement **selectAccountById** and the IN body should contain the account id we want to retrieve, such as an **Integer** type.

We can do the same for some of the other operations, such as **QueryForList**:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-ibatis/src/test/java/org/apache/camel/component/ibatis/IbatisQueryForListTest.java}
```

And the same for **UPDATE**, where we can send an **Account** object as IN body to iBatis:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-ibatis/src/test/java/org/apache/camel/component/ibatis/IbatisQueryForUpdateTest.java}
```

Scheduled polling example

Since this component does not support scheduled polling, you need to use another mechanism for triggering the scheduled polls, such as the Timer or Quartz components.

In the sample below we poll the database, every 30 seconds using the Timer component and send the data to the JMS queue:

```
java from("timer://pollTheDatabase?delay=30000").to("ibatis:selectAllAccounts?")
```

```
statementType=QueryForList".to("activemq:queue:allAccounts");
```

And the iBatis SQL map file used:

```
xml <!-- Select with no parameters using the result map for Account class. --> <select id="selectAllAccounts" resultMap="AccountResult"> select * from ACCOUNT </select>
```

Using onConsume

This component supports executing statements **after** data have been consumed and processed by Camel. This allows you to do post updates in the database. Notice all statements must be **UPDATE** statements. Camel supports executing multiple statements whose name should be separated by comma.

The route below illustrates we execute the **consumeAccount** statement data is processed. This allows us to change the status of the row in the database to processed, so we avoid consuming it twice or more.

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-
ibatis/src/test/java/org/apache/camel/component/ibatis/IBatisQueueTest.java}
And the statements in the sqlmap file:
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-
ibatis/src/test/resources/org/apache/camel/component/ibatis/Account.xml}
{snippet:id=e2|lang=xml|url=camel/trunk/components/camel-
ibatis/src/test/resources/org/apache/camel/component/ibatis/Account.xml}
Endpoint See Also
```

- MyBatis

IRC COMPONENT

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

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

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

URI format

```
irc:nick@host[:port]/#room[?options]
irc:nick@host[:port]channels=#channel1,#channel2,#channel3[?options]
```

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

Options

Name	Description	Example	Default Value
channels	Comma separated list of IRC channels to join.	channels=#channel1,#channel2	null
nickname	The nickname used in chat.	irc:nick@irc.server.org#channel1 or irc:irc.server.org#channel1?	null
username	The IRC server user name.	irc:nick@irc.server.org#channel1 or irc:irc.server.org#channel1?	Same as nickname.
password	The IRC server password.	password=complex	None
realname	The IRC user's actual name.	realname=myname	None
nickPassword	Camel 2.18: Your IRC server nickname password.	nickPassword=mystic	None
colors	Whether or not the server supports color codes.	true, false	true
onempty	Whether or not the server returns responses to commands or informational messages.	true, false	false
onnick	Handle user nick events.	true, false	true
onquit	Handle user quit events.	true, false	true
onjoin	Handle user join events.	true, false	true
onkick	Handle kick events.	true, false	true
onmode	Handle mode change events.	true, false	true
onPart	Handle user part events.	true, false	true
onTopic	Handle topic change events.	true, false	true
onPrivmsg	Handle private message events.	true, false	true
trustManager	The trust manager used to verify the SSL server's certificate.	trustManager=#referenceToTrustManagerBean	the default trust manager, which accepts all certificates, will be used
keys	Camel 2.2: Comma separated list of IRC channel keys. Important to be listed in same order as channels. When joining multiple channels, the keys will be inserted into the Registry. This reference overrides any keys defined at the component level. See Using the JSSE Configuration Utility. Note that this setting overrides #mySslContextParameters	irc:myNick@irc.server.org/#channel1? keys=channelkey1,channelkey2	null
sslContextParameters	configured SslContextParameters at the component level. See Using the JSSE Configuration Utility. Note that this setting overrides the trustManager option.	#mySslContextParameters	null

SSL Support

Using the JSSE Configuration Utility

As of Camel 2.9, the IRC component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the IRC component.

Programmatic configuration of the endpoint

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/truststore.jks");
ksp.setPassword("keystorePassword");

TrustManagersParameters tmp = new TrustManagersParameters();
```

```

tmp.setKeyStore(ksp);
SSLContextParameters scp = new SSLContextParameters();
scp.setTrustManagers(tmp);
Registry registry = ...
registry.bind("sslContextParameters", scp);
...
from(...)
    .to("ircs://camel-prd-user@server:6669/#camel-test?nickname=camel-prd&password=password&sslContextParameters=#sslContextParameters");

```

Spring DSL based configuration of endpoint

```

...
<camel:sslContextParameters
    id="sslContextParameters">
    <camel:trustManager>
        <camel:keyStore
            resource="/users/home/server/truststore.jks"
            password="keystorePassword"/>
    </camel:keyManagers>
</camel:sslContextParameters>...
...
<to uri="ircs://camel-prd-user@server:6669/#camel-test?nickname=camel-prd&password=password&sslContextParameters=#sslContextParameters"/>...

```

Using the legacy basic configuration options

You can also connect to an SSL enabled IRC server, as follows:

```
ircs:host[:port]/#room?username=user&password=pass
```

By default, the IRC transport uses SSLDefaultTrustManager. If you need to provide your own custom trust manager, use the trustManager parameter as follows:

```
ircs:host[:port]/#room?username=user&password=pass&trustManager=#referenceToMyTrustManagerBean
```

Using keys

Available as of Camel 2.2

Some irc rooms requires you to provide a key to be able to join that channel. The key is just a secret word.

For example we join 3 channels where as only channel 1 and 3 uses a key.

```
irc:nick@irc.server.org?channels=#chan1,#chan2,#chan3&keys=chan1key,,chan3key
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JASYPT COMPONENT

Available as of Camel 2.5

Jasypt is a simplified encryption library which makes encryption and decryption easy. Camel integrates with Jasypt to allow sensitive information in Properties files to be encrypted. By dropping camel-jasypt on the classpath those encrypted values will automatically be decrypted on-the-fly by Camel. This ensures that human eyes can't easily spot sensitive information such as usernames and passwords.

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

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

Tooling

The Jasypt component provides a little command line tooling to encrypt or decrypt values.

The console output the syntax and which options it provides:

Apache Camel Jasypt takes the following options -h or -help = Displays the help screen -c or -command <command> = Command either encrypt or decrypt -p or -password <password> = Password to use -i or -input <input> = Text to encrypt or

decrypt -a or -algorithm <algorithm> = Optional algorithm to use

For example to encrypt the value `tiger` you run with the following parameters. In the apache camel kit, you cd into the lib folder and run the following java cmd, where <CAMEL_HOME> is where you have downloaded and extract the Camel distribution.

```
$ cd <CAMEL_HOME>/lib $ java -jar camel-jasypt-2.5.0.jar -c encrypt -p secret -i tiger
```

Which outputs the following result

Encrypted text: qaEEacuW7BUTi8LcMgyjKw==

This means the encrypted representation `qaEEacuW7BUTi8LcMgyjKw==` can be decrypted back to `tiger` if you know the master password which was `secret`.

If you run the tool again then the encrypted value will return a different result. But decrypting the value will always return the correct original value.

So you can test it by running the tooling using the following parameters:

```
$ cd <CAMEL_HOME>/lib $ java -jar camel-jasypt-2.5.0.jar -c decrypt -p secret -i qaEEacuW7BUTi8LcMgyjKw==
```

Which outputs the following result:

Decrypted text: tiger

The idea is then to use those encrypted values in your Properties files. Notice how the password value is encrypted and the value has the tokens surrounding `ENC(value here)`

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-jasypt/src/test/resources/org/apache/camel/component/jasypt/myproperties.properties}
```

Tooling dependencies for Camel 2.5 and 2.6

The tooling requires the following JARs in the classpath, which has been enlisted in the `MANIFEST.MF` file of `camel-jasypt` with `optional/` as prefix. Hence why the java cmd above can pickup the needed JARs from the Apache Distribution in the optional directory.

`jasypt-1.6.jar commons-lang-2.4.jar commons-codec-1.4.jar icu4j-4.0.1.jar` Java 1.5 users

The `icu4j-4.0.1.jar` is only needed when running on JDK 1.5.

This JAR is not distributed by Apache Camel and you have to download it manually and copy it to the `lib/optional` directory of the Camel distribution.

You can download it from Apache Central Maven repo.

Tooling dependencies for Camel 2.7 or better

Jasypt 1.7 onwards is now fully standalone so no additional JARs is needed.

URI Options

The options below are exclusive for the Jasypt component.

confluenceTableSmall

Name	Default Value	Type	Description
password	null	String	Specifies the master password to use for decrypting. This option is mandatory. See below for more details.
algorithm	null	String	Name of an optional algorithm to use.

Protecting the master password

The master password used by Jasypt must be provided, so that it's capable of decrypting the values. However having this master password out in the open may not be an ideal solution. Therefore you could for example provide it as a JVM system property or as a OS environment setting. If you decide to do so then the `password` option supports prefixes which dictates this. `sysenv:` means to lookup the OS system environment with the given key. `sys:` means to lookup a JVM system property.

For example you could provided the password before you start the application

```
$ export CAMEL_ENCRYPTION_PASSWORD=secret
```

Then start the application, such as running the start script.

When the application is up and running you can unset the environment

```
$ unset CAMEL_ENCRYPTION_PASSWORD
```

The password option is then a matter of defining as follows: `password=sysenv:CAMEL_ENCRYPTION_PASSWORD`.

Example with Java DSL

In Java DSL you need to configure Jasypt as a `JasyptPropertiesParser` instance and set it on the Properties component as show below:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-jasypt/src/test/java/org/apache/camel/component/jasypt/JasyptPropertiesTest.java}
```

The properties file `myproperties.properties` then contain the encrypted value, such as shown below. Notice how the password value is encrypted and the value has the tokens surrounding `ENC(value here)`

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-jasypt/src/test/resources/org/apache/camel/component/jasypt/myproperties.properties}
```

Example with Spring XML

In Spring XML you need to configure the `JasyptPropertiesParser` which is shown below. Then the Camel Properties component is told to use jasypt as the properties parser, which means Jasypt has its chance to decrypt values looked up in the properties.

```
<!-- define the jasypt properties parser with the given password to be used --> <bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser"> <property name="password" value="secret"/> </bean> <!-- define the camel properties component --> <bean id="properties" class="org.apache.camel.component.properties.PropertiesComponent"> <!-- the properties file is in the classpath --> <property name="location" value="classpath:org/apache/camel/component/jasypt/myproperties.properties"/> <!-- and let it leverage the jasypt parser --> <property name="propertiesParser" ref="jasypt"/> </bean>
The Properties component can also be inlined inside the <camelContext> tag which is shown below. Notice how we use the propertiesParserRef attribute to refer to Jasypt.
<!-- define the jasypt properties parser with the given password to be used --> <bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser"> <!-- password is mandatory, you can prefix it with sysenv: or sys: to indicate it should use an OS environment or JVM system property value, so you dont have the master password defined here --> <property name="password" value="secret"/> </bean> <camelContext xmlns="http://camel.apache.org/schema/spring">
<!-- define the camel properties placeholder, and let it leverage jasypt --> <propertyPlaceholder id="properties" location="classpath:org/apache/camel/component/jasypt/myproperties.properties" propertiesParserRef="jasypt"/> <route> <from uri="direct:start"/> <to uri="{{cool.result}}"/> </route> </camelContext>
```

Example with Blueprint XML

In Blueprint XML you need to configure the `JasyptPropertiesParser` which is shown below. Then the Camel Properties component is told to use jasypt as the properties parser, which means Jasypt has its chance to decrypt values looked up in the properties.

```
xml-blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0" xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd" <cm:property-placeholder id="myblue" persistent-id="mypersistent"> <!-- list some properties for this test --> <cm:default-properties> <cm:property name="cool.result" value="mock:{{cool.password}}"/> <cm:property name="cool.password" value="ENC(bsW9uV37gQ0QHFU7KO03Ww==)"/> </cm:default-properties> </cm:property-placeholder> <!-- define the jasypt properties parser with the given password to be used --> <bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser"> <property name="password" value="secret"/> </bean>
<camelContext xmlns="http://camel.apache.org/schema/blueprint"> <!-- define the camel properties placeholder, and let it leverage jasypt --> <propertyPlaceholder id="properties" location="blueprint:myblue" propertiesParserRef="jasypt"/> <route> <from uri="direct:start"/> <to uri="{{cool.result}}"/> </route> </camelContext> </blueprint>
The Properties component can also be inlined inside the <camelContext> tag which is shown below. Notice how we use the propertiesParserRef attribute to refer to Jasypt.
xml-blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0" xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd" <!-- define the jasypt properties parser with the given password to be used --> <bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser"> <property name="password" value="secret"/> </bean>
<camelContext xmlns="http://camel.apache.org/schema/blueprint"> <!-- define the camel properties placeholder, and let it leverage jasypt --> <propertyPlaceholder id="properties" location="classpath:org/apache/camel/component/jasypt/myproperties.properties" propertiesParserRef="jasypt"/> <route> <from uri="direct:start"/> <to uri="{{cool.result}}"/> </route> </camelContext> </blueprint>
```

See Also

- Security
- Properties
- Encrypted passwords in ActiveMQ - ActiveMQ has a similar feature as this `camel-jasypt` component

JAVASPACE COMPONENT

Available as of Camel 2.1

The `Javaspace` component is a transport for working with any JavaSpace compliant implementation and 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.net.jini.core.entry.Entry` class. It is also possible to pass the bean ID of a template that can be used for reading/taking the entries from the space.

This component can be used for sending/receiving any serializable object acting as a sort of generic transport. The JavaSpace

component contains a special optimization for dealing with the BeanExchange. It can be used to invoke a POJO remotely, using a JavaSpace as a transport.

This latter feature can provide a simple implementation of the master/worker pattern, where a POJO provides the business logic for the worker.

Look at the test cases for examples of various use cases for this component.

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

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

URI format

```
javaspaces:jini://host[?options]
```

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

Options

Name	Default Value	Description
spaceName	null	Specifies the JavaSpace name.
verb	take	Specifies the verb for getting JavaSpace entries. The values can be: take or read.
transactional	false	If true, sending and receiving entries is performed within a transaction.
transactionalTimeout	Long.MAX_VALUE	Specifies the transaction timeout.
concurrentConsumers	1	Specifies the number of concurrent consumers getting entries from the JavaSpace.
templateId	null	If present, this option specifies the Space template ID of the template to use for reading/taking entries.

Examples

Sending and Receiving Entries

```
// sending route
from("direct:input")
.to("javaspaces:jini://localhost?spaceName=mySpace");

// receiving Route
from("javaspaces:jini://localhost?spaceName=mySpace&templateId=template&verb=take&concurrentConsumers=1")
.to("mock:foo");
```

In this case the payload can be any object that inherits from the Jini Entry type.

Sending and receiving serializable objects

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

Using JavaSpace as a remote invocation transport

The JavaSpace component has been tailored to work in combination with the Camel bean component. It is therefore possible to call a remote POJO using JavaSpace as the transport:

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

// server side
from("javaspaces:jini://localhost?concurrentConsumers=10&spaceName=mySpace")
.to("mock:foo");
```

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

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JBI COMPONENT

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

See below for information about how to use StreamSource types from ServiceMix in Camel.
The following code:

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

Automatically exposes a new endpoint to the bus, where the service QName is {<http://foo.bar.org>}MyService and the endpoint name is MyEndpoint (see #URI-format).

When a JBI endpoint appears at the end of a route, for example:

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

The messages sent by this producer endpoint are sent to the already deployed JBI endpoint.

URI format

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

The separator that should be used in the endpoint URL is:

- / (forward slash), if serviceNamespace starts with <http://>, or
- : (colon), if serviceNamespace starts with <urn:>

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

Using the **jbi:service:** or **jbi:endpoint:** URI formats sets the service QName on the JBI endpoint to the one specified. Otherwise, the default Camel JBI Service QName is used, which is:

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

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

Examples

```
jbi:service:http://foo.bar.org/MyService
jbi:endpoint:urn:foo:bar:MyService:MyEndpoint
jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint
jbi:name:cheese
```

URI options

Name	Default value	Description
mep	MEP of the Camel Exchange	Allows users to override the MEP set on the Exchange object. Valid values for this option are in-only, in-out, robust-in-out and in-optional-out.
operation	Value of the jbi.operation header property	Specifies the JBI operation for the MessageExchange. If no value is supplied, the JBI binding will use the value of the jbi.operation header property.
serialization	basic	Default value (basic) will check if headers are serializable by looking at the type, setting this option to strict will detect objects that can not be serialized although they implement the Serializable interface. Set to nocheck to disable this check altogether, note that this should only be used for in-memory transports like SEDAFlow, otherwise you can expect to get NotSerializableException thrown at runtime.
convertException	false	false: send any exceptions thrown from the Camel route back unmodified true: convert all exceptions to a JBI FaultException (can be used to avoid non-serializable exceptions or to implement generic error handling)

Examples

```
jbi:service:http://foo.bar.org/MyService?mep=in-out      (override the MEP, use InOut JBI MessageExchanges)
jbi:endpoint:urn:foo:bar:MyService:MyEndpoint?mep=in    (override the MEP, use InOnly JBI MessageExchanges)
jbi:endpoint:urn:foo:bar:MyService:MyEndpoint?operation=(http://www.mycompany.org)AddNumbers
(override the operation for the JBI Exchange to (http://www.mycompany.org)AddNumbers)
```

Using Stream bodies

If you are using a stream type as the message body, you should be aware that a stream is only capable of being read once. So if you enable DEBUG logging, the body is usually logged and thus read. To deal with this, Camel has a `streamCaching` option that can cache the stream, enabling you to read it multiple times.

```
from("jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint").streamCaching().to("xslt:transform.xsl", "bean:doSomething");
```

The stream caching is default enabled, so it is not necessary to set the `streamCaching()` option.

We store big input streams (by default, over 64K) in a temp file using `cachedOutputStream`. When you close the input stream, the temp file will be deleted.

Creating a JBI Service Unit

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

If you have an existing Maven project that you need to convert into a JBI Service Unit, you may want to consult ServiceMix Maven JBI Plugins for further help. The key steps are as follows:

- Create a Spring XML file at `src/main/resources/camel-context.xml` to bootstrap your routes inside the JBI Service Unit.
- Change the POM file's packaging to `jbi-service-unit`.

Your `pom.xml` should look something like this to enable the `jbi-service-unit` packaging:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/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>x.x.x</camel-version>
    <servicemix-version>3.3</servicemix-version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.servicemix</groupId>
      <artifactId>servicemix-camel</artifactId>
      <version>${servicemix-version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.servicemix</groupId>
      <artifactId>servicemix-core</artifactId>
      <version>${servicemix-version}</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <defaultGoal>install</defaultGoal>

    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <!-- creates the JBI deployment unit -->
      <plugin>
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <version>${servicemix-version}</version>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- ServiceMix Camel module
- Using Camel with ServiceMix
- Cookbook on using Camel with ServiceMix

JCR COMPONENT

The `jcr` component allows you to add/read nodes to/from a JCR compliant content repository (for example, Apache Jackrabbit) with its producer, or register an EventListener with the consumer.

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

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

URI format

```
jcr://user:password@repository/path/to/node
```

Consumer added

From **Camel 2.10** onwards you can use consumer as an EventListener in JCR or a producer to read a node by identifier.

Usage

The repository element of the URI is used to look up the JCR Repository object in the Camel context registry.

Producer

Name	Default Value	Description
<code>CamelJcrOperation</code>	<code>CamelJcrInsert</code>	CamelJcrInsert or CamelJcrGetByld.operation to use
<code>CamelJcrNodeName</code>	<code>null</code>	Used to determine the node name to use
<code>CamelJcrPrimaryType</code>	<code>null</code>	Camel 2.16: To use to specify primary node type when creating adding a new node

When a message is sent to a JCR producer endpoint:

- If the operation is CamelJcrInsert: A new node is created in the content repository, all the message headers of the IN message are transformed to `javax.jcr.value` instances and added to the new node and the node's UUID is returned in the OUT message.
- If the operation is CamelJcrGetByld: A new node is retrieved from the repository using the message body as node identifier.

Please note that the JCR Producer used message properties instead of message headers in Camel versions earlier than 2.12.3. See <https://issues.apache.org/jira/browse/CAMEL-7067> for more details.

Consumer

The consumer will connect to JCR periodically and return a `List<javax.jcr.observation.Event>` in the message body.

Name	Default Value	Description
<code>eventTypes</code>	<code>0</code>	A combination of one or more event types encoded as a bit mask value such as <code>javax.jcr.observation.Event.NODE_ADDED</code> , <code>javax.jcr.observation.Event.NODE_REMOVED</code> , etc.
<code>deep</code>	<code>false</code>	When it is true, events whose associated parent node is at current path or within its subgraph are received.
<code>uids</code>	<code>null</code>	Only events whose associated parent node has one of the identifiers in the comma separated uid list will be received.
<code>nodeTypeNames</code>	<code>null</code>	Only events whose associated parent node has one of the node types in the subgraph of the current node. If in this list will be received.
<code>nodeTypeNames</code>	<code>false</code>	Only events whose associated parent node has one of the node types in the subgraph of the current node. If in this list will be received.
<code>sessionLiveCheckInterval</code>	<code>8000</code>	interval in milliseconds to wait before each session live checking.
<code>sessionLiveCheckIntervalStart</code>	<code>3000</code>	interval in milliseconds to wait before the first session live checking.
<code>username</code>	<code>null</code>	Camel 2.15: Allows to specify the username as a uri parameter instead of in the authority section of the uri
<code>password</code>	<code>null</code>	Camel 2.15: Allows to specify the password as a uri parameter instead of in the authority section of the uri
<code>workspaceName</code>	<code>null</code>	Camel 2.16: Allows to specify a workspace different from default

Example

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

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

The following code will register an EventListener under the path import-application/inbox for Event.NODE_ADDED and Event.NODE_REMOVED events (event types 1 and 2, both masked as 3) and listening deep for all the children.

```
<route>
    <from uri="jcr://user:pass@repository/import-application/inbox?eventTypes=3&deep=true" />
    <to uri="direct:execute-import-application" />
</route>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

JDBC COMPONENT

The **jdbc** component enables you to access databases through JDBC, where SQL queries (SELECT) and operations (INSERT, UPDATE, etc) are sent in the message body. This component uses the standard JDBC API, unlike the SQL Component component, which uses spring-jdbc.

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

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

This component can only be used to define producer endpoints, which means that you cannot use the JDBC component in a from() statement.

URI format

jdbc:dataSourceName[?options]

This component only supports producer endpoints.

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

Options

confluenceTableSmall

Name	Default Value	Description
readsize	0	The default maximum number of rows that can be read by a polling query. The default value is 0.
statement.<xxx>	null	Camel 2.1: Sets additional options on the java.sql.Statement that is used behind the scenes to execute the queries. For instance, statement.setMaxRows=10. For detailed documentation, see the java.sql.Statement javadoc documentation.
useJDBC4ColumnNameAndLabelSemantics	true	Camel 2.2: Sets whether to use JDBC 4/3 column label/name semantics. You can use this option to turn it false in case you have issues with your JDBC driver to select data. This only applies when using SQL SELECT using aliases (e.g. SQL SELECT id as identifier, name as given_name from persons).
resetAutoCommit	true	Camel 2.9: If true, Camel will set the autoCommit on the JDBC connection to be false, commit the change after executing the statement and reset the autoCommit flag of the connection at the end. If the JDBC connection does not support resetting the autoCommit flag, set this to false. When used with XA transactions you most likely need to set it to false so that the transaction manager is in charge of committing this tx.
allowNamedParameters	true	Camel 2.12: Whether to allow using named parameters in the queries
prepareStatementStrategy		Camel 2.12: Allows to plugin to use a custom org.apache.camel.component.jdbc.JdbcPreparedStatementStrategy

		to control preparation of the query and prepared statement.
useHeadersAsParameters	false	<p>Camel 2.12: Set this option to true to use the <code>prepareStatementStrategy</code> with named parameters. This allows to define queries with named placeholders, and use headers with the dynamic values for the query placeholders.</p>
outputType	SelectList	<p>Camel 2.12.1: outputType='SelectList', for consumer or producer, will output a List of Map. <code>SelectOne</code> will output single Java object in the following way:</p> <ul style="list-style-type: none"> a) If the query has only single column, then that JDBC Column object returned. (such as <code>SELECT COUNT(*) FROM PROJECT</code> will return a Long object). b) If the query has more than one column, then it will return a Map of t result. c) If the outputClass is set, then it will convert the query result into an Java bean object by calling all the setters that match the column name. It will assume your class has a default constructor to create instance v From Camel 2.14 onwards then <code>SelectList</code> is also supported. d) If the query resulted in more than one rows, it throws an non-unique result exception. <p>Camel 2.14.0: New <code>StreamList</code> output type value that streams the results of the query using an <code>Iterator<Map<String, Object>></code>, it can be used along with the Splitter EIP.</p>
outputClass	null	<p>Camel 2.12.1: Specify the full package and class name to use as conversion when <code>outputType=SelectOne</code>. From Camel 2.14 onwards then <code>SelectList</code> is also supported.</p>
beanRowMapper		<p>Camel 2.12.1: To use a custom <code>org.apache.camel.component.jdbc.BeanRowMapper</code> when using <code>outputClass</code>. The default implementation will lower case the row name and skip underscores, and dashes. For example "CUST_ID" is mapped as "custId".</p>
useGetBytesForBlob	false	<p>Camel 2.16: To read BLOB columns as bytes instead of string data. This may be needed for certain databases such as Oracle where you must read BLOB columns as bytes.</p>

Result

By default the result is returned in the OUT body as an `ArrayList<HashMap<String, Object>>`. The List object contains the list of rows and the Map objects contain each row with the String key as the column name. You can use the option `outputType` to control the result.

Note: This component fetches `ResultSetMetaData` to be able to return the column name as the key in the Map.

Message Headers

Header	Description
<code>CamelJdbcRowCount</code>	If the query is a SELECT, query the row count is returned in this OUT header.
<code>CamelJdbcUpdateCount</code>	If the query is an UPDATE, query the update count is returned in this OUT header.
<code>CamelGeneratedkeysRows</code>	Camel 2.10: Rows that contains the generated keys.
<code>CamelGeneratedkeysRowCount</code>	Camel 2.10: The number of rows in the header that contains generated keys.
<code>CamelJdbcColumnNames</code>	Camel 2.11.1: The column names from the ResultSet as a <code>java.util.Set</code> type.
<code>CamelJdbcParameters</code>	Camel 2.12: A <code>java.util.Map</code> which has the headers to be used if <code>useHeadersAsParameters</code> has been enabled.

Generated keys

Available as of Camel 2.10

If you insert data using SQL INSERT, then the RDBMS may support auto generated keys. You can instruct the JDBC producer to return the generated keys in headers.

To do that set the header `CamelRetrieveGeneratedKeys=true`. Then the generated keys will be provided as headers with the keys listed in the table above.

You can see more details in this unit test.

Using generated keys does not work with together with named parameters.

Using named parameters

Available as of Camel 2.12

In the given route below, we want to get all the projects from the projects table. Notice the SQL query has 2 named parameters, `:?lic` and `:?min`.
 Camel will then lookup these parameters from the message headers. Notice in the example above we set two headers with constant value for the named parameters:
`from("direct:projects") .setHeader("lic", constant("ASF")) .setHeader("min", constant(123)) .setBody("select * from projects where license = :?lic and id > :?min order by id") .to("jdbc:myDataSource?useHeadersAsParameters=true")`
 You can also store the header values in a `java.util.Map` and store the map on the headers with the key `camelJdbcParameters`.

Samples

In the following example, we fetch the rows from the customer table.

First we register our datasource in the Camel registry as `testdb`:

```
{snippet:id=register|lang=java|url=camel/trunk/components/camel-jdbc/src/test/java/org/apache/camel/component/jdbc/AbstractJdbcTestSupport.java}Then we configure a route that routes to the JDBC component, so the SQL will be executed. Note how we refer to the testdb datasource that was bound in the previous step:
```

```
{snippet:id=route|lang=java|url=camel/trunk/components/camel-jdbc/src/test/java/org/apache/camel/component/jdbc/JdbcRouteTest.java}Or you can create a DataSource in Spring like this:
```

```
{snippet:id=example|lang=java|url=camel/trunk/components/camel-jdbc/src/test/resources/org/apache/camel/component/jdbc/camelContext.xml}We create an endpoint, add the SQL query to the body of the IN message, and then send the exchange. The result of the query is returned in the OUT body:
```

```
{snippet:id=invoke|lang=java|url=camel/trunk/components/camel-jdbc/src/test/java/org/apache/camel/component/jdbc/JdbcRouteTest.java}If you want to work on the rows one by one instead of the entire ResultSet at once you need to use the Splitter EIP such as:
```

In Camel 2.13.x or older(`snippet:id=e1|lang=java|url=camel/trunk/components/camel-jdbc/src/test/java/org/apache/camel/component/jdbc/JdbcRouteSplitTest.java`)In Camel 2.14.x or newer
`from("direct:hello") // here we split the data from the testdb into new messages one by one // so the mock endpoint will receive a message per row in the table // the StreamList option allows to stream the result of the query without creating a List of rows // and notice we also enable streaming mode on the splitter .to("jdbc:testdb?outputType=StreamList") .split(body()) .streaming() .to("mock:result");`

Sample - Polling the database every minute

If we want to poll a database using the JDBC component, we need to combine it with a polling scheduler such as the Timer or Quartz etc. In the following example, we retrieve data from the database every 60 seconds:

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

Sample - Move Data Between Data Sources

A common use case is to query for data, process it and move it to another data source (ETL operations). In the following example, we retrieve new customer records from the source table every hour, filter/transform them and move them to a destination table:

```
javafrom("timer://MoveNewCustomersEveryHour?period=3600000") .setBody(constant("select * from customer where create_time > (sysdate-1/24)")) .to("jdbc:testdb") .split(body()) .process(new MyCustomerProcessor()) //filter/transform results as needed .setBody(simple("insert into processed_customer values('${body[ID]}','${body[NAME]}'')")) .to("jdbc:testdb");
```

Endpoint See Also

- [SQL](#)

JETTY COMPONENT

The producer is deprecated - do not use. We only recommend using jetty as consumer (eg from jetty)

The `jetty` component provides HTTP-based endpoints for consuming and producing HTTP requests. That is, the Jetty component behaves as a simple Web server. Jetty can also be used as an HTTP client which means you can also use it with Camel as a producer.

Stream

The `assert` call appears in this example, because the code is part of an unit test. Jetty is stream based, which means the input it receives is submitted to Camel as a stream. That means you will only be able to read the content of the stream **once**.

If you find a situation where the message body appears to be empty or you need to access the `Exchange.HTTP_RESPONSE_CODE` data multiple times, e.g., doing multi-casting, or re-delivery error handling, you should use Stream caching or convert the message body to a `String` which is safe to be re-read multiple times.

Maven users should add the following dependency to their pom.xml to use this component:

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

URI Format

jetty:[http://hostname\[:port\]/\[resourceUri\]\[?options\]](http://hostname[:port]/[resourceUri][?options])

Query options should be appended to the URI using the following format: ?option=value&option=value&...

Options

confluenceTableSmall

Option	Default Value	Description
bridgeEndpoint	false	<p>Camel 2.1: If the option is true, HttpProducer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for request. You may also set the throwExceptionOnFailure to be false to let the HttpProducer send all the fault response back.</p> <p>Camel 2.3: If the option is true, HttpProducer and camelServlet will skip the gzip processing if the Content-Encoding is gzip.</p> <p>Consider setting disableStreamCache=true to optimize when bridging.</p>
chunked	true	<p>Camel 2.2: If this option is false Jetty Servlet will disable the HTTP streaming and set the Content-Length header on the response</p> <p>Camel 2.6: Allows to set a timeout in milliseconds when using Jetty as consumer (server). By default Jetty uses 30000. You can use a value of <= 0 to never expire. If a timeout occurs then the request will be expired and Jetty will return back an HTTP error 503 to the client.</p>
continuationTimeout	null	<p>This option is only in use when using Jetty with the Asynchronous Routing Engine.</p>
cookieHandler	null	<p>Camel 2.19: Producer only Configure a cookie handler to maintain a HTTP session.</p>
disableStreamCache	false	<p>Camel 2.3: Determines whether or not the raw input stream from Jetty is cached or not (Camel will read the stream into a in memory/overflow to file, Stream caching) cache. By default Camel will cache the Jetty input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However, you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store.</p> <p>DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times. If you use Jetty to bridge/proxy an endpoint then consider enabling this option to improve performance, in case you do not need to read the message payload multiple times.</p>
enableCORS	false	<p>Camel 2.15: if the option is true, Jetty server will setup the CrossOriginFilter which supports the CORS out of box.</p>
enableJmx	false	<p>Camel 2.3: If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.</p>
enableMulti-partFilter	true	<p>Camel 2.5: Whether Jetty org.eclipse.jetty.servlets.multi-partFilter is enabled or not.</p> <p>Set this option to false when bridging endpoints, to ensure multi-part requests is proxied/bridged as well.</p>
filterInit.xxx	null	<p>Camel 2.17: Configuration for the InitParameters of filter.</p> <p>For example, setting filterInit.parameter=value the parameter could be used when calling the filter init() method.</p>
filtersRef	null	<p>Camel 2.9: Allows using a custom filters which is putted into a list and can be find in the Registry</p>
handlers	null	<p>Specifies a comma-delimited set of org.mortbay.jetty.Handler instances in your Registry (such as your Spring ApplicationContext). These handlers are added to the Jetty Servlet context (for example, to add security).</p> <p>Note: you can not use different handlers with different Jetty endpoints using the same port number. The handlers is associated to the port number. If you need different handlers, then use different port numbers.</p>
headerFilterStrategy	null	<p>Camel 2.11: Reference to a instance of org.apache.camel.spi.HeaderFilterStrategy in the Registry. It will be used to apply the custom headerFilterStrategy on the new create HttpJettyEndpoint.</p>

httpBindingRef	null	Reference to an org.apache.camel.component.http.HttpBinding in the Registry. HttpBinding can be used to customize how a response should be written for the consumer.
httpClient.xxx	null	Configuration of Jetty's HttpClient. For example, setting <code>httpClient.idleTimeout=30000</code> sets the idle timeout to 30 seconds. And <code>httpClient.timeout=30000</code> sets the request timeout to 30 seconds, in case you want to timeout sooner if you have long running request/response calls.
httpClient	null	To use a shared org.eclipse.jetty.client.HttpClient for all producers created by this endpoint. This option should only be used in special circumstances.
httpClientMinThreads	null	Camel 2.11: Producer only: To set a value for minimum number of threads in HttpClient thread pool. This setting override any setting configured on component level. Notice that both a min and max size must be configured. If not set it default to min 8 threads used in Jetty's thread pool.
httpClientMaxThreads	null	Camel 2.11: Producer only: To set a value for maximum number of threads in HttpClient thread pool. This setting override any setting configured on component level. Notice that both a min and max size must be configured. If not set it default to max 16 threads used in Jetty's thread pool.
httpMethodRestrict	null	Camel 2.11: Consumer only: Used to only allow consuming if the HttpMethod matches, such as GET/POST/PUT etc. From Camel 2.15: multiple methods can be specified separated by comma.
jettyHttpBindingRef	null	Camel 2.6.0+: Reference to an org.apache.camel.component.jetty.JettyHttpBinding in the Registry. JettyHttpBinding can be used to customize how a response should be written for the producer.
matchOnUriPrefix	false	Whether or not the CamelServlet should try to find a target consumer by matching the URI prefix if no exact match is found. See here How do I let Jetty match wildcards .
multi-partFilterRef	null	Camel 2.6: Allows using a custom multi-part filter. Note: setting multi-partFilterRef forces the value of enablemulti-partFilter to true.
okStatusCodeRange	200-299	Camel 2.16: Producer only The status codes which is considered a success response. The values are inclusive. The range must be defined as from-to with the dash included.
optionsEnabled	false	Camel 2.17: Specifies whether to enable HTTP OPTIONS for this Jetty consumer. By default OPTIONS is turned off.
proxyHost	null	Camel 2.11: Producer only The HTTP proxy Host URL which will be used by Jetty client.
proxyPort	null	Camel 2.11: Producer only The HTTP proxy port which will be used by Jetty client.
responseBufferSize	null	Camel 2.12: To use a custom buffer size on the javax.servlet.ServletResponse.
sendDateHeader	false	Camel 2.14: if the option is true, jetty server will send the date header to the client which sends the request. Note: ensure that there are no any other camel-jetty endpoints that share the same port, otherwise this option may not work as expected.
sendServerVersion	true	Camel 2.13: if the option is true, jetty will send the server header with the jetty version information to the client which sends the request. Note: ensure that there are no any other camel-jetty endpoints that share the same port, otherwise this option may not work as expected.
sessionSupport	false	Specifies whether to enable the session manager on the server side of Jetty.
sslContextParameters	null	Camel 2.17: Reference to a org.apache.camel.util.jsse.SSLContextParameters in the Registry. This reference overrides any configured SSLContextParameters at the component level. See Using the JSSE Configuration Utility.
sslContextParametersRef	null	Camel 2.8: Deprecated Reference to a org.apache.camel.util.jsse.SSLContextParameters in the Registry. This reference overrides any configured SSLContextParameters at the component level. See Using the JSSE Configuration Utility.
throwExceptionOnFailure	true	Option to disable throwing the <code>HttpoperationFailedException</code> in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.

traceEnabled	false	Specifies whether to enable HTTP TRACE for this Jetty consumer. By default TRACE is turned off.
transferException	false	<p>Camel 2.6: If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type.</p> <p>On the producer side the exception will be deserialized and thrown as is, instead of the HttpoperationFailedException. The caused exception is required to be serialized.</p>
urlRewrite	null	<p>Camel 2.11: Producer only Refers to a custom org.apache.camel.component.http.UrlRewrite which allows you to rewrite URLs when you bridge/proxy endpoints.</p> <p>See more details at UrlRewrite and How to use Camel as a HTTP proxy between a client and server.</p>
useContinuation	true	Camel 2.6: Whether or not to use Jetty continuations for the Jetty Server.

Message Headers

Camel uses the same message headers as the HTTP component. From **Camel 2.2**, it also uses (`Exchange.HTTP CHUNKED`, `CamelHttpchunked`) header to toggle chunked encoding on the camel-jetty consumer. Camel also populates all `request.parameter` and `request.headers`. For example, given a client request with the URL, `http://myserver/myserver?orderid=123`, the exchange will contain a header named `orderid` with the value 123.

From **Camel 2.2.0**: you can get the `request.parameter` from the message header not only from GET HTTP Method, but also other HTTP method.

Usage

The Jetty component supports both consumer and producer endpoints. Another option for producing to other HTTP endpoints, is to use the HTTP Component

Component Options

The JettyHttpComponent provides the following options:
confluenceTableSmall

Option	Default Value	Description
allowJavaSerializedObject	false	<p>Camel 2.16.1/2.15.5: Whether to allow java serialization when a request uses context-type=application/x-java-serialized-object.</p> <p>When true, be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.</p>
enableJmx	false	Camel 2.3: If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.
errorHandler	null	Camel 2.15: This option is used to set the <code>ErrorHandler</code> that Jetty server uses.
httpClient	null	<p>Deprecated: Producer only: To use a custom HttpClient with the jetty producer.</p> <p>Note: from Camel 2.11 this option has been removed. Set the option on the endpoint instead.</p>
httpClientMaxThreads	null	Producer only: To set a value for maximum number of threads in HttpClient thread pool. Notice that both a min and max size must be configured.
httpClientMinThreads	null	Producer only: To set a value for minimum number of threads in HttpClient thread pool. Notice that both a min and max size must be configured.
httpClientThreadPool	null	<p>Deprecated: Producer only: To use a custom thread pool for the client.</p> <p>Note: this option has been removed from Camel 2.11.</p>
maxThreads	null	Camel 2.5 Consumer only: To set a value for maximum number of threads in server thread pool. Notice that both a min and max size must be configured.
minThreads	null	Camel 2.5 Consumer only: To set a value for minimum number of threads in server thread pool. Notice that both a min and max size must be configured.
proxyHost	null	Camel 2.12.2/2.11.3 To use an HTTP proxy.
proxyPort	null	Camel 2.12.2/2.11.3: To use an HTTP proxy.

socketConnectors	null	Camel 2.5 Consumer only: A map which contains per port number specific HTTP connectors. Uses the same principle as <code>sslSocketConnectors</code> and therefore see section SSL support for more details.
socketConnectorProperties	null	Camel 2.5 Consumer only: A map which contains general HTTP connector properties. Uses the same principle as <code>sslSocketConnectorProperties</code> and therefore see section SSL support for more details.
sslContextParameters	null	Camel 2.8: To configure a custom SSL/TLS configuration options at the component level. See Using the JSSE Configuration Utility for more details.
sslKeyPassword	null	Consumer only: The password for the keystore when using SSL.
sslKeystore	null	Consumer only: The path to the keystore.
sslPassword	null	Consumer only: The password when using SSL.
sslSocketConnectors	null	Camel 2.3 Consumer only: A map which contains per port number specific SSL connectors. See section SSL support for more details.
sslSocketConnectorProperties	null	Camel 2.5 Consumer only: A map which contains general SSL connector properties. See section SSL support for more details.
requestBufferSize	null	Camel 2.11.2: Allows to configure a custom value of the request buffer size on the Jetty connectors.
requestHeaderSize	null	Camel 2.11.2: Allows to configure a custom value of the request header size on the Jetty connectors.
responseBufferSize	null	Camel 2.11.2: Allows to configure a custom value of the response buffer size on the Jetty connectors.
responseHeaderSize	null	Camel 2.11.2: Allows to configure a custom value of the response header size on the Jetty connectors.
threadPool	null	Camel 2.5 Consumer only: To use a custom thread pool for the server. This option should only be used in special circumstances.

Producer Example

The following is a basic example of how to send an HTTP request to an existing HTTP endpoint.

Java DSL:
`javafrom("direct:start").to("jetty://http://www.google.com");`

XML DSL:
`xml<route> <from uri="direct:start"/> <to uri="jetty://http://www.google.com"/> </route>`

Consumer Example

In this sample we define a route that exposes a HTTP service at `http://localhost:8080/myapp/myservice`:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-jetty9/src/test/java/org/apache/camel/component/jetty/JettyRouteTest.java}
```

Usage of localhost

When you specify `localhost` in a URL, Camel exposes the endpoint only on the local TCP/IP network interface, so it cannot be accessed from outside the machine it operates on.

If you need to expose a Jetty endpoint on a specific network interface, the numerical IP address of this interface should be used as the host. If you need to expose a Jetty endpoint on all network interfaces, the `0.0.0.0` address should be used.

To listen across an entire URI prefix, see [How do I let Jetty match wildcards](#).

If you actually want to expose routes by HTTP and already have a Servlet, you should instead refer to the [Servlet Transport](#).

Our business logic is implemented in the `MyBookService` class, which accesses the HTTP request contents and then returns a response.

Note: The `assert` call appears in this example, because the code is part of an unit test.

```
{snippet:id=e2|lang=java|url=camel/trunk/components/camel-jetty9/src/test/java/org/apache/camel/component/jetty/JettyRouteTest.java}The following sample shows a content-based route that routes all requests containing the URI parameter, one, to the endpoint, mock:one, and all others to mock:other.{snippet:id=e1|lang=java|url=camel/trunk/components/camel-jetty9/src/test/java/org/apache/camel/component/jetty/JettyContentBasedRouteTest.java}If a client sends an HTTP request, http://serverUri?one=hello, the Jetty component will copy the HTTP request parameter, one to the exchange's in.header. We can then use the simple language to route exchanges that contain this header to a specific endpoint and all others to another. If we used a language more powerful than Simple, e.g., EL or OGNL, then we can also test for the parameter value and route based on the header value as well.
```

Session Support

The session support option, `sessionSupport`, can be used to enable a `HttpSession` object and access the session object while processing the exchange.

For example, the following route enables sessions:

```
xml<route> <from uri="jetty:http://0.0.0.0/myapp/myservice/?sessionSupport=true"/> <processRef ref="myCode"/> </route>
The myCode Processor can be instantiated by a Spring bean element:
xml<bean id="myCode" class="com.mycompany.MyCodeProcessor"/>
Where the processor implementation can access the HttpSession as follows:
javapublic void process(Exchange exchange) throws Exception { HttpSession session =
exchange.getIn(HttpMessage.class).getRequest().getSession(); // ... }
```

SSL Support (HTTPS)

Using the JSSE Configuration Utility

From **Camel 2.8**: the `camel-jetty` component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the Jetty component.

Programmatic configuration of the component

```
javaKeyStoreParameters ksp = new KeyStoreParameters(); ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword"); KeyManagersParameters kmp = new KeyManagersParameters(); kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword"); SSLContextParameters scp = new SSLContextParameters(); scp.setKeyManagers(kmp);
JettyComponent jettyComponent = getContext().getComponent("jetty", JettyComponent.class);
jettyComponent.setSslContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
xml<camel:sslContextParameters id="sslContextParameters"> <camel:keyManagers keyPassword="keyPassword">
<camel:keyStore resource="/users/home/server/keystore.jks" password="keystorePassword"/> </camel:keyManagers>
</camel:sslContextParameters> <o uri="jetty:https://127.0.0.1/mail/?sslContextParametersRef=sslContextParameters"/>
```

Configuring Jetty Directly

Jetty provides SSL support out of the box. To enable Jetty to run in SSL mode, simply format the URI using the `https://` prefix.

Example:

```
xml<from uri="jetty:https://0.0.0.0/myapp/myservice"/>
```

Jetty also needs to know where to load your keystore from and what passwords to use in order to load the correct SSL certificate. Set the following JVM System Properties:

Before Camel 2.3:

confluenceTableSmall

Property	Description
<code>jetty.ssl.keystore</code>	Specifies the location of the Java keystore file, which contains the Jetty server's own x.509 certificate in a <i>key entry</i> . A key entry stores the x.509 certificate (effectively, the <i>public key</i>) and also its associated private key.
<code>jetty.ssl.password</code>	The store password, which is required to access the keystore file (this is the same password that is supplied to the keystore command's <code>-storepass</code> option).
<code>jetty.ssl.keypassword</code>	The key password, which is used to access the certificate's key entry in the keystore (this is the same password that is supplied to the keystore command's <code>-keypass</code> option).

From Camel 2.3:

confluenceTableSmall

Property	Description
<code>org.eclipse.jetty.ssl.keystore</code>	Specifies the location of the Java keystore file, which contains the Jetty server's own x.509 certificate in a <i>key entry</i> . A key entry stores the x.509 certificate (effectively, the <i>public key</i>) and also its associated private key.
<code>org.eclipse.jetty.ssl.password</code>	The store password, which is required to access the keystore file (this is the same password that is supplied to the keystore command's <code>keystore</code> option).
<code>org.eclipse.jetty.ssl.keypassword</code>	The key password, which is used to access the certificate's key entry in the keystore (this is the same password that is supplied to the keystore command's <code>keystore</code> option).

For details of how to configure SSL on a Jetty endpoint, read the following Jetty documentation. Some SSL properties aren't exposed directly by Camel. However, Camel does expose the underlying `sslSocketConnector`, which will allow you to set properties like `needClientAuth` for mutual authentication requiring a client certificate or `wantClientAuth` for mutual authentication where a client doesn't need a certificate but can have one.

There's a slight difference between the various Camel versions:

Up to Camel 2.2

```
xml<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent"> <property name="sslSocketConnectors">
<map> <entry key="8043"> <bean class="org.mortbay.jetty.security.SslSocketConnector"> <property
name="password" value="..."/> <property name="keyPassword" value="..."/> <property name="keystore" value="..."/> <property
name="needClientAuth" value="..."/> <property name="truststore" value="..."/> </bean> </entry> </map> </property> </bean>
```

Camel 2.3, 2.4

```
xml<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent"> <property name="sslSocketConnectors">
<map> <entry key="8043"> <bean class="org.eclipse.jetty.server.ssl.SslSocketConnector"> <property
name="password" value="..."/> <property name="keyPassword" value="..."/> <property name="keystore" value="..."/> <property
name="needClientAuth" value="..."/> <property name="truststore" value="..."/> </bean> </entry> </map> </property> </bean>
```

From **Camel 2.5**: we switch to use `sslSelectChannelConnector` *

```
xml<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent"> <property name="sslSocketConnectors">
<map> <entry key="8043"> <bean class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector"> <property
name="password" value="..."/> <property name="keyPassword" value="..."/> <property name="keystore" value="..."/> <property
name="needClientAuth" value="..."/> <property name="truststore" value="..."/> </bean> </entry> </map> </property> </bean>
```

The value you use as keys in the above map is the port you configure Jetty to listen on.

Configuring General SSL Properties

From **Camel 2.5**: instead of a per port number specific SSL socket connector (as shown above) you can now configure general properties which applies for all SSL socket connectors (which is not explicit configured as above with the port number as entry).

```
xml<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent"> <property
name="sslSocketConnectorProperties"> <map> <entry key="password" value="..."/> <entry key="keyPassword" value="..."/>
<entry key="keystore" value="..."/> <entry key="needClientAuth" value="..."/> <entry key="truststore" value="..."/> </map>
</property> </bean>
```

How to Obtain A Reference to the X509Certificate

Jetty stores a reference to the certificate in the `HttpServletRequest` which you can access from code as follows:

```
javaHttpServletRequest req = exchange.getIn().getBody(HttpServletRequest.class); X509Certificate cert = (X509Certificate)
req.getAttribute("javax.servlet.request.X509Certificate")
```

Configuring General HTTP Properties

From **Camel 2.5**: instead of a per port number specific HTTP socket connector (as shown above) you can now configure general properties which applies for all HTTP socket connectors (which is not explicit configured as above with the port number as entry).

```
xml<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent"> <property
name="socketConnectorProperties"> <map> <entry key="acceptors" value="4"/> <entry key="maxIdleTime" value="300000"/>
</map> </property> </bean>
```

How to Get the Value of The X-Forwarded-For HTTP Header Using `HttpServletRequest.getRemoteAddr()`

If the HTTP requests are handled by an Apache server and forwarded to Jetty with `mod_proxy`, the original client IP address is in the `X-Forwarded-For` header and the `HttpServletRequest.getRemoteAddr()` will return the address of the Apache proxy.

Jetty has a forwarded property which takes the value from `X-Forwarded-For` and places it in the `HttpServletRequest.getRemoteAddr()` property. This property is not available directly through the endpoint configuration but it can be easily added using the `socketConnectors` property:

```
xml<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent"> <property name="socketConnectors">
<map> <entry key="8080"> <bean class="org.eclipse.jetty.server.nio.SelectChannelConnector"> <property name="forwarded"
value="true"/> </bean> </entry> </map> </property> </bean>
```

This is particularly useful when an existing Apache server handles TLS connections for a domain and proxies them to application servers internally.

Default Behavior for Returning HTTP Status Codes

The default behavior of HTTP status codes is defined by the `org.apache.camel.component.http.DefaultHttpBinding` class, which handles how a response is written and also sets the HTTP status code. If the exchange was processed successfully, the 200 HTTP status code is returned. If the exchange failed with an exception, the 500 HTTP status code is returned, and the stacktrace is returned in the body. If you want to specify which HTTP status code to return, set the code in the `Exchange.HTTP_RESPONSE_CODE` header of the OUT message.

Customizing `HttpBinding`

By default, Camel uses the `org.apache.camel.component.http.DefaultHttpBinding` to handle how a response is written. If you like, you can customize this behavior either by implementing your own `HttpBinding` class or by extending `DefaultHttpBinding` and overriding the appropriate methods.

The following example shows how to customize the `DefaultHttpBinding` in order to change how exceptions are returned:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-}
```

jetty9/src/test/java/org/apache/camel/component/jetty/HttpBindingRefTest.java} We can then create an instance of our binding and register it in the Spring registry as follows:

```
xml<bean id="mybinding" class="com.mycompany.MyHttpBinding"/>
And then we can reference this binding when we define the route:
xml<route> <from uri="jetty:http://0.0.0.0:8080/myapp/myservice?httpBindingRef=mybinding"/> <to uri="bean:doSomething"/>
</route>
```

Jetty Handlers and Security Configuration

You can configure a list of Jetty handlers on the endpoint, which can be useful for enabling advanced Jetty security features. These handlers are configured in Spring XML as follows:

```
xml<-- Jetty Security handling --> <bean id="userRealm" class="org.mortbay.jetty.plus.jaas.JAASUserRealm"> <property name="name" value="tracker-users"/> <property name="loginModuleName" value="ldaploginmodule"/> </bean> <bean id="constraint" class="org.mortbay.jetty.security.Constraint"> <property name="name" value="BASIC"/> <property name="roles" value="tracker-users"/> <property name="authenticate" value="true"/> </bean> <bean id="constraintMapping" class="org.mortbay.jetty.security.ConstraintMapping"> <property name="constraint" ref="constraint"/> <property name="pathSpec" value="/"/> </bean> <bean id="securityHandler" class="org.mortbay.jetty.security.SecurityHandler"> <property name="UserRealm" ref="userRealm"/> <property name="constraintMappings" ref="constraintMapping"/> </bean>
```

From **Camel 2.3**: you can configure a list of Jetty handlers as follows:

```
xml<-- Jetty Security handling --> <bean id="constraint" class="org.eclipse.jetty.http.security.Constraint"> <property name="name" value="BASIC"/> <property name="roles" value="tracker-users"/> <property name="authenticate" value="true"/> </bean> <bean id="constraintMapping" class="org.eclipse.jetty.security.ConstraintMapping"> <property name="constraint" ref="constraint"/> <property name="pathSpec" value="/"/> </bean> <bean id="securityHandler" class="org.eclipse.jetty.security.ConstraintSecurityHandler"> <property name="authenticator" ref="BasicAuthenticator"/> <property name="constraintMappings" ref="constraintMapping"/> </list>
```

You can then define the endpoint as:

```
javafrom("jetty:http://0.0.0.0:9080/myservice?handlers=securityHandler")
```

If you need more handlers, set the `handlers` option equal to a comma-separated list of bean IDs.

How to Customize the Response on an HTTP 500 Server Error

You may want to return a custom reply message when something goes wrong, instead of the default reply message Camel Jetty replies with. You could use a custom `HttpBinding` to be in control of the message mapping, but often it may be easier to use Camel's Exception Clause to construct the custom reply message.

Example: return the message: dude something went wrong for the HTTP error code 500:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-jetty9/src/test/java/org/apache/camel/component/jetty/JettyOnExceptionHandledTest.java}
```

Multi-Part Form Support

From **Camel 2.3.0**: camel-jetty support to multi-part form post out of box. The submitted form-data are mapped into the message header. camel-jetty creates an attachment for each uploaded file. The file name is mapped to the name of the attachment. The content type is set as the content type of the attachment file name. You can find the example here.

```
{snippet:id=e1|lang=java|url=camel/components/camel-jetty9/src/test/java/org/apache/camel/component/jetty/MultiPartFormTest.java|title=Note: getName() functions as shown below in versions 2.5 and higher. In earlier versions you receive the temporary file name for the attachment instead}
```

Jetty JMX Support

From **Camel 2.3.0**: camel-jetty supports the enabling of Jetty's JMX capabilities at the component and endpoint level with the endpoint configuration taking priority.

Note: JMX must be enabled within the Camel context in order to enable JMX support in this component as the component provides Jetty with a reference to the MBeanServer registered with the Camel context.

As the camel-jetty component caches and reuses Jetty resources for a given protocol/host/port pairing, this configuration option will only be evaluated during the creation of the first endpoint to use a protocol/host/port pairing.

Example: given two routes created from the following XML fragments, JMX support would remain enabled for all endpoints listening on: `https://0.0.0.0`.

```
xml<from uri="jetty:https://0.0.0.0/myapp/myservice1?enableJmx=true"/> xml<from uri="jetty:https://0.0.0.0/myapp/myservice2?enableJmx=false"/>
```

The camel-jetty component also provides for direct configuration of the Jetty MBeanContainer. Jetty creates MBean names dynamically. If you are running another instance of Jetty outside of the Camel context and sharing the same MBeanContainer between the instances, you can provide both instances with a reference to the same MBeanContainer in order to avoid name collisions when registering Jetty MBeans.

Endpoint See Also

- HTTP

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

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

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

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

URI format Camel 2.15 or older

```
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	References the XML file foo/bar.rng on the classpath
rnc: http://foo.com/bar.rnc	References the RelaxNG Compact Syntax file from the URL, http://foo.com/bar.rnc

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

URI format Camel 2.16

```
jing:someLocalOrRemoteResource
```

From Camel 2.16 the component use **jing** as name, and you can use the option **compactSyntax** to turn on either RNG or RNC mode.

Options

Option	Default	Description
compactSyntax	false	Whether to validate using RelaxNG compact syntax or not.

Example

The following example shows how to configure a route from the endpoint **direct:start** which then goes to one of two endpoints, either **mock:valid** or **mock:invalid** based on whether or not the XML matches the given RelaxNG Compact Syntax schema (which is supplied on the classpath).

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <doTry>
      <to uri="jing:org/apache/camel/component/validator/jing/schema.rnc?compactSyntax=true"/>
      <doCatch>
        <exception>org.apache.camel.ValidationException</exception>
        <to uri="mock:valid"/>
      </doCatch>
      <doFinally>
        <to uri="mock:finally"/>
      </doFinally>
    </doTry>
  </route>
</camelContext>
```

See Also

- Configuring Camel
- Component
- Endpoint

- Getting Started

JMS COMPONENT

Using ActiveMQ

If you are using Apache ActiveMQ, you should prefer the ActiveMQ component as it has been optimized for ActiveMQ. All of the options and samples on this page are also valid for the ActiveMQ component.

Transacted and caching

See section *Transactions and Cache Levels* below if you are using transactions with JMS as it can impact performance.

Request/Reply over JMS

Make sure to read the section *Request-reply over JMS* further below on this page for important notes about request/reply, as Camel offers a number of options to configure for performance, and clustered environments.

This component allows messages to be sent to (or consumed from) a JMS Queue or Topic. It uses Spring's JMS support for declarative transactions, including Spring's `JmsTemplate` for sending and a `MessageListenerContainer` for consuming.

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

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

URI Format

`jms:[queue:]topic:]destinationName[?options]`

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

`jms:FOO.BAR`

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

`jms:queue:FOO.BAR`

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

`jms:topic:Stocks.Prices`

You append query options to the URI using the following format: `?option=value&option=value&...`

Notes

Using ActiveMQ

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

If you intend 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 optimized to use ActiveMQ efficiently
- Use the `PoolingConnectionFactory` in ActiveMQ

Transactions and Cache Levels

`transactionCacheLevel`

If you are consuming messages and using transactions (`transacted=true`) then the default cache level can negatively impact performance. If you are using XA transactions then you cannot cache as it can cause the XA transaction to not work properly.

If you are *not* using XA, then you should consider caching as it speeds up performance, such as setting `cacheLevelName=CACHE_CONSUMER`. Through Camel 2.7.x, the default setting for `cacheLevelName` is `CACHE_CONSUMER`. You will need to explicitly set `cacheLevelName=CACHE_NONE`. In Camel 2.8 onward, the default setting for `cacheLevelName` is `CACHE_AUTO`. This default auto detects the mode and sets the cache level accordingly to:

- `CACHE_CONSUMER` when `transacted=false`
- `CACHE_NONE` when `transacted=true`

So you can say the default setting is conservative. Consider using `cacheLevelName=CACHE_CONSUMER` if you are using non-XA transactions.

Durable Subscriptions

If you wish to use durable topic subscriptions, you need to specify both `clientId` and `durableSubscriptionName`. 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.

Message Header Mapping

When using message headers, the JMS specification states that header names must be valid Java identifiers. So try to name your headers to be valid Java identifiers. One benefit of doing this is that you can then use your headers inside a JMS Selector

(whose SQL92 syntax mandates Java identifier syntax for headers).

A simple strategy for mapping header names is used by default. The strategy is to replace any dots and hyphens in the header name as shown below and to reverse the replacement when the header name is restored from a JMS message sent over the wire. What does this mean? No more losing method names to invoke on a bean component, no more losing the filename header for the File Component, and so on.

The current header name strategy for accepting header names in Camel is:

- Dots are replaced by `_DOT_` and the replacement is reversed when Camel consume the message
- Hyphen is replaced by `_HYPHEN_` and the replacement is reversed when Camel consumes the message

Configuration Options

You can configure many different properties on the JMS endpoint which map to properties on the `JMSConfiguration` POJO. Mapping to Spring JMS

Many of these properties map to properties on Spring JMS, which Camel uses for sending and receiving messages. Therefore for more information about these properties consult the Spring documentation.

The options are divided into two tables, the first one contains the most common options. The second table contains the less common and more advanced options.

Common Options

confluenceTableSmall

Option	Default Value	Description
<code>clientId</code>	<code>null</code>	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. It is typically only required for durable topic subscriptions. You may prefer to use Virtual Topics instead.
<code>concurrentConsumers</code>	1	<p>Specifies the default number of concurrent consumers.</p> <p>From Camel 2.10.3: this option can also be used when doing request/reply over JMS.</p> <p>From Camel 2.16: there is a new <code>replyToConcurrentConsumers</code>.</p> <p>See the <code>maxMessagesPerTask</code> option to control dynamic scaling up/down of threads.</p> <p>When using ActiveMQ beware that the default prefetch policy loads 1000 messages per consumer. See What is the prefetch limit on how to change this.</p>
<code>disableReplyTo</code>	<code>false</code>	When true, a producer will behave like a <code>InOnly</code> exchange with the exception that <code>JMSReplyTo</code> header is sent out and not be suppressed like in the case of <code>InOnly</code> . Like <code>InOnly</code> the producer will not wait for a reply. A consumer with this flag will behave like <code>InOnly</code> . This feature can be used to bridge <code>Inout</code> requests to another queue so that a route on the other queue will send its response directly back to the original <code>JMSReplyTo</code> .
<code>durableSubscriptionName</code>	<code>null</code>	The durable subscriber name for specifying durable topic subscriptions. The <code>clientId</code> option must be configured as well.
<code>maxConcurrentConsumers</code>	1	<p>Specifies the maximum number of concurrent consumers.</p> <p>From Camel 2.10.3: this option can also be used when doing request/reply over JMS.</p> <p>From Camel 2.16: there is a new <code>replyToMaxConcurrentConsumers</code>.</p> <p>See also the <code>maxMessagesPerTask</code> option to control dynamic scaling up/down of threads.</p> <p>The <code>maxMessagesPerTask</code> option must be set to an integer greater than 0 for threads to scale down. Otherwise, the number of threads will remain at <code>maxConcurrentConsumers</code> until shutdown.</p>
<code>maxMessagesPerTask</code>	-1	The number of messages a task can receive after which it's terminated. The default, -1, is unlimited.
<code>preserveMessageQos</code>	<code>false</code>	If you use a range for concurrent consumers e.g., <code>concurrentConsumers < maxConcurrentConsumers</code> then this option can be used to set a value to e.g., 100 to control how fast the consumers will shrink when less work is required.

		Set to <code>true</code> , if you want to send message using the QoS settings specified on the message, instead of the QoS settings on the JMS endpoint. The following headers are considered:
		<ul style="list-style-type: none"> • <code>JMSPriority</code> • <code>JMSDeliveryMode</code> • <code>JMSExpiration</code>.
		You can provide some or all of them. If not provided, Camel will fall back to use the values from the endpoint instead. So, when using this option, the headers override the values from the endpoint.
		The <code>explicitQosEnabled</code> option, by contrast, will only use options set on the endpoint, and not values from the message header.
<code>replyTo</code>	<code>null</code>	Provides an explicit <code>ReplyTo</code> destination, which overrides any incoming value of <code>Message.getJMSReplyTo()</code> . If you do Request Reply over JMS then make sure to read the section <i>Request-reply over JMS</i> further below for more details, and the <code>replyToType</code> option as well.
<code>replyToConcurrentConsumers</code>	<code>1</code>	Camel 2.16: Specifies the default number of concurrent consumers when doing request/reply over JMS.
<code>replyToMaxConcurrentConsumers</code>	<code>1</code>	Camel 2.16: Specifies the maximum number of concurrent consumers when doing request/reply over JMS. See the <code>maxMessagesPerTask</code> option to control dynamic scaling up/down of threads.
<code>replyToOnTimeoutMaxConcurrentConsumers</code>	<code>1</code>	Camel 2.17.2: Specifies the maximum number of concurrent consumers for continue routing when timeout occurred when using request/reply over JMS.
<code>replyToOverride</code>	<code>null</code>	Camel 2.15: Provides an explicit <code>ReplyTo</code> destination in the JMS message, which overrides the setting of <code>ReplyTo</code> . It is useful if you want to forward the message to a remote Queue and receive the reply message from the <code>ReplyTo</code> destination.
<code>replyToType</code>	<code>null</code>	Camel 2.9: Allows for explicitly specifying which kind of strategy to use for <code>ReplyTo</code> queues when doing request/reply over JMS. Possible values are: <ul style="list-style-type: none"> • <code>Temporary</code> • <code>Shared</code> • <code>Exclusive</code> By default Camel will use <code>Temporary</code> queues. However if <code>replyTo</code> has been configured, then <code>shared</code> is used by default. This option allows you to use exclusive queues instead of shared queues. For more details see below, and especially the notes about the implications if running in a clustered environment, and the fact that shared reply queues has lower performance than its alternatives <code>Temporary</code> and <code>Exclusive</code> .
<code>requestTimeout</code>	<code>20000</code>	Producer only: The timeout for waiting for a reply when using the Inout Exchange Pattern (in milliseconds). From Camel 2.13/2.12.3 : you can include the header <code>CamelJmsRequestTimeout</code> to override this endpoint configured timeout value, and thus have per message individual timeout values. See below in section <i>About time to live</i> for more details. See also the <code>requestTimeoutCheckerInterval</code> option.
<code>selector</code>	<code>null</code>	Sets the JMS Selector, which is an SQL 92 predicate that is used to filter messages within the broker. You may have to encode special characters like '=' as %3D. Before Camel 2.3.0 : this option was not supported in <code>CamelConsumerTemplate</code> .
<code>timeToLive</code>	<code>null</code>	When sending messages, specifies the time-to-live of the message (in milliseconds). See below in section <i>About time to live</i> for more details.
<code>transacted</code>	<code>false</code>	Specifies whether to use transacted mode for sending/receiving messages using the <code>Inonly</code> Exchange Pattern.

testConnectionOnStartup	false	Camel 2.1: Specifies whether to test the connection on startup. This ensures that when Camel starts that all the JMS consumers have a valid connection to the JMS broker. If a connection cannot be granted then Camel throws an exception on startup. This ensures that Camel is not started with failed connections. From Camel 2.8: also the JMS producers is tested as well.
-------------------------	-------	---

Advanced Options

confluenceTableSmall

Option	Default Value	Description
acceptMessageswhileStopping	false	Specifies whether the consumer accept messages while stopping. You may consider enabling this option, if you still want messages enqueued on the queue. If this configuration is enabled, messages may be rejected, and the JMS broker may be rejected, and eventually the message is lost. To avoid this scenario it's recommended this option is disabled.
acknowledgementModeName	AUTO_ACKNOWLEDGE	The JMS acknowledgement name, which is one of: <ul style="list-style-type: none">• SESSION_TRANSACTED• CLIENT_ACKNOWLEDGE• AUTO_ACKNOWLEDGE• DUPS_OK_ACKNOWLEDGE.
acknowledgementMode	-1	The JMS acknowledgement mode defined as a string corresponding to the acknowledgement mode. For the regular mode, use the acknowledgementModeName instead.
allowNullBody	true	Camel 2.9.3/2.10.1: Whether to allow sending a message with null body, then an JMSEXCEPTION is thrown.
allowReplyManagerQuickstop	false	Whether the defaultMessageListener container for messaging allows the defaultMessageListener link. JmsConfiguration isAcceptMessageswhileStopped. This quick stop ability is enable for reply managers you must enable this flag.
alwaysCopyMessage	false	If true, Camel will always make a JMS message producer for sending. Copying the message is required if replyTo is set to destinationSelectorName. Camel will set the alwaysCopyMessage=true.
asyncConsumer	false	Camel 2.9: Whether the JmsConsumer process JmsConsumer may pickup the next message from the queue processed asynchronously (by the AsynchronousConsumer). Note: if transactions has been enabled, then the transactions must be executed synchronously (otherwise the consumer will be disabled).
asyncStartListener	false	Camel 2.10: Whether to startup the JmsConsumer route. For example if a JmsConsumer cannot get a connection while retrying and/or failover. This will cause Camel to start the route. Camel will let routes startup, while the JmsConsumer dedicated thread in asynchronous mode. If this is not established, then an exception is logged when trying to receive messages. You can then restart the route.
asyncStopListener	false	Camel 2.10: Whether to stop the JmsConsumer route.
autoStartup	true	Specifies whether the consumer container should automatically start up.
cacheLevel		Sets the cache level by ID for the underlying JMS provider details.
cacheLevelName	<ul style="list-style-type: none"> • CACHE_AUTO (Camel >= 2.8.0) • CACHE_CONSUMER (Camel <= 2.7.1) 	Sets the cache level by name for the underlying JMS provider. <ul style="list-style-type: none"> • CACHE_NONE • CACHE_CONNECTION • CACHE_SESSION • CACHE_CONSUMER • CACHE_AUTO

From **Camel 2.8:** the default is CACHE_AUTO.

		For Camel 2.7.1 and older the default is <code>cac</code> . See the Spring documentation and Transactional JMS for more information.								
connectionFactory	null	The default JMS connection factory to use for the <code>templateConnectionFactory</code> , if neither is specified.								
consumerType	Default	<p>The Spring JMS listener type to use. A valid value is one of:</p> <table border="1"> <thead> <tr> <th>consumerType</th> <th>Spring JMS Listener</th> </tr> </thead> <tbody> <tr> <td>Default</td> <td><code>org.springframework.jms.listener.DefaultMessageListenerContainer</code></td> </tr> <tr> <td>Simple</td> <td><code>org.springframework.jms.listener.SimpleMessageListenerContainer</code></td> </tr> <tr> <td>Custom</td> <td>From Camel 2.10.2: The <code>messageListenerContainer</code> bean to use.</td> </tr> </tbody> </table> <p>This option was temporarily removed in Camel 2.12.2/2.13. It is now available again.</p> <p>Camel 2.10.4: Specifies what default <code>TaskExecutor</code> to use for <code>DefaultMessageListenerContainer</code>, for both producer endpoints. Possible values: <code>SimpleAsync</code> or <code>ThreadPool</code> (uses Spring's <code>ThreadPoolTaskExecutor</code>). If not set, it defaults to the previous behavior of <code>SimpleAsync</code> for reply consumer endpoints and <code>SimpleAsync</code> for reply consumer endpoints.</p> <p>The use of <code>ThreadPool</code> is recommended to dynamically increasing and decreasing concurrent consumers.</p>	consumerType	Spring JMS Listener	Default	<code>org.springframework.jms.listener.DefaultMessageListenerContainer</code>	Simple	<code>org.springframework.jms.listener.SimpleMessageListenerContainer</code>	Custom	From Camel 2.10.2: The <code>messageListenerContainer</code> bean to use.
consumerType	Spring JMS Listener									
Default	<code>org.springframework.jms.listener.DefaultMessageListenerContainer</code>									
Simple	<code>org.springframework.jms.listener.SimpleMessageListenerContainer</code>									
Custom	From Camel 2.10.2: The <code>messageListenerContainer</code> bean to use.									
defaultTaskExecutorType	SimpleAsync	<p>Camel 2.12.2/2.13: Specifies the delivery mode to use for <code>javax.jms.deliveryMode</code>.</p> <p>Specifies whether persistent delivery is used by the consumer.</p> <p>Specifies the JMS Destination object to use on the consumer endpoint.</p> <p>Specifies the JMS destination name to use on the consumer endpoint.</p> <p>A pluggable <code>org.springframework.jms.support.destination.DestinationResolver</code> to let you to use your own resolver (for example, to look up destinations in a database).</p>								
deliveryMode	null	Camel 2.8: Use this option to force disabling the delivery mode of the JMS, then Camel will by default use the request/reply pattern. The problem is that the sender and receiver are in sync. This is not always so easy to do. You can also not set a time to live value on the sent message in the system.								
deliveryPersistent	true	See below in section <i>About time to live</i> for more information.								
destination	null	Enables eager loading of JMS properties as soon as possible, which is very inefficient, because the JMS properties might not be used for testing purposes, to ensure JMS properties are loaded correctly.								
destinationName	null	Camel 2.8.2, 2.9: Specifies a <code>org.springframework.jms.support.destination.DestinationResolver</code> to handle any uncaught exceptions thrown while processing the message.								
destinationResolver	null	By default these exceptions will be logged as errors.								
disableTimeToLive	false	From Camel 2.9.1: you can configure logging of the below two options. This makes it much easier to log errors from the <code>ErrorHandler</code> .								
eagerLoadingOfProperties	false	Camel 2.9.1: Configures the logging level at <code>WARN</code> .								
errorHandler	null	Camel 2.9.1: Controls whether a stacktrace should be printed when an exception occurs.								
errorHandlerLoggingLevel	WARN	Specifies the JMS Exception Listener that is to be used.								
errorHandlerLogStackTrace	true	Set if the <code>deliveryMode</code> , <code>priority</code> or <code>timeToLive</code> messages. This option is based on Spring's <code>JmsTemplate</code> . The <code>timeToLive</code> options are applied to the current option, which operates at message granularity, not at message headers.								
exceptionListener	null	Specifies whether the listener session should be closed when an exception occurs.								
explicitQosEnabled	false	Camel 2.7: When using <code>mapJmsMessage=false</code> on a new JMS destination if you touch the headers (e.g. <code>headers.put("foo", "bar")</code>) then Camel will send the original JMS message.								
exposeListenerSession	true	Camel 2.8.2, 2.9: Specify the limit for the number of idle consumer sessions.								
forceSendOriginalMessage	false	Specifies the limit for idle executions of a receiver.								
idleConsumerLimit	1									
idleTaskExecutionLimit	1									

<code>includeSentJMSMessageID</code>	<code>false</code>	execution. If this limit is reached, the task will skip the case of dynamic scheduling; see the <code>maxConcurrentConsumers</code> available from Spring.						
<code>includeAllJMSXProperties</code>	<code>false</code>	<p>Camel 2.10.3: Only applicable when sending to JMS. Enabling this option will enrich the Camel Exchange with JMS client when the message was sent to the JMS provider.</p> <p>Camel 2.11.2/2.12: Whether to include all JMSX properties in the Message. When set to <code>true</code> properties such as <code>JMSXGroupID</code>, <code>JMSXDeliveryCount</code> etc. are included.</p> <p>Note: If you are using a custom header <code>Foo</code> then Camel will ignore it.</p>						
<code>jmsKeyFormatStrategy</code>	<code>default</code>	<p>Pluggable strategy for encoding and decoding header keys.</p> <table border="1"> <thead> <tr> <th>Strategy</th><th>Description</th></tr> </thead> <tbody> <tr> <td><code>default</code></td><td>Safely marshals dots and hyphens in header keys.</td></tr> <tr> <td><code>passthrough</code></td><td>Leaves the key as is. Appropriate for header keys containing illegal characters.</td></tr> </tbody> </table> <p>Note: optionally, a custom implementation can be created by implementing <code>org.apache.camel.component.jms.JmsKeyFormatStrategy</code>.</p> <p>Allows you to force the use of a specific javax.jms.Message type. Possible values are:</p> <ul style="list-style-type: none"> • Bytes • Map • Object • Stream • Text 	Strategy	Description	<code>default</code>	Safely marshals dots and hyphens in header keys.	<code>passthrough</code>	Leaves the key as is. Appropriate for header keys containing illegal characters.
Strategy	Description							
<code>default</code>	Safely marshals dots and hyphens in header keys.							
<code>passthrough</code>	Leaves the key as is. Appropriate for header keys containing illegal characters.							
<code>jmsMessageType</code>	<code>null</code>	By default Camel determines which JMS message type to use. You can override the default behavior.						
<code>jmsOperations</code>	<code>null</code>	Allows you to use your own implementation of the <code>org.apache.camel.component.jms.JmsOperations</code> interface. Camel uses <code>JmsTemplate</code> by default. See the JMS API docs .						
<code>lazyCreateTransactionManager</code>	<code>true</code>	If <code>true</code> , Camel will create a <code>JmsTransactionManager</code> when option <code>transacted=true</code> .						
<code>listenerConnectionFactory</code>	<code>null</code>	The JMS connection factory used for consuming messages.						
<code>mapJmsMessage</code>	<code>true</code>	Specifies whether Camel should auto map the JMS message type to Java objects. Such as <code>javax.jms.TextMessage</code> to a <code>java.util.List</code> . See the message type mapping works.						
<code>maximumBrowseSize</code>	<code>-1</code>	Limits the number of messages fetched at most.						
<code>messageConverter</code>	<code>null</code>	To use a custom Spring <code>org.springframework.core.convert.converter.Converter</code> converter, Camel can be 100% in control how to map to/from a JMS message.						
<code>messageIdEnabled</code>	<code>true</code>	When sending, specifies whether message IDs should be generated.						
<code>messageListenerContainerFactoryRef</code>	<code>null</code>	Camel 2.10.2: Registry ID of the <code>org.apache.camel.component.jms.listener.MessageListenerContainerFactory</code> implementation. Setting this will automatically set consumer properties.						
<code>messageTimestampEnabled</code>	<code>true</code>	Specifies whether time-stamps should be enabled.						
<code>password</code>	<code>null</code>	The password for the connector factory.						
<code>priority</code>	<code>4</code>	Values greater than 1 specify the message priority higher than the highest). The <code>explicitQosEnabled</code> option must be <code>true</code> for this effect.						
<code>pubsubNoLocal</code>	<code>false</code>	Specifies whether to inhibit the delivery of messages to the local subscriber.						
<code>receiveTimeout</code>	<code>1000</code>	The timeout for receiving messages (in milliseconds).						
<code>recoveryInterval</code>	<code>5000</code>	Specifies the interval between recovery attempts in milliseconds. The default is 5000 ms.						
<code>replyToSameDestinationAllowed</code>	<code>false</code>	Camel 2.16: Consumer only: Whether a JMS consumer can reply to the same destination that the consumer is using to receive messages. This is useful for consumers that are consuming and sending back the same message.						
<code>replyToCacheLevelName</code>	<code>CACHE_CONSUMER</code>	Camel 2.9.1: Sets the cache level by name for this option only applies when using fixed reply <code>org.apache.camel.component.jms.JmsConsumer</code> for exclusive or shared w/o reply selector. Some JMS brokers require this parameter to be set to <code>CACHE_NONE</code> in order to support it.						

		Note: The value CACHE_NONE cannot be used. CACHE_CONSUMER or CACHE_SESSION, must be used.
replyToDestinationSelectorName	null	Sets the JMS Selector using the fixed name to others when using a shared queue (that is, if you specify a selector on the queue).
replyToDeliveryPersistent	true	Specifies whether to use persistent delivery by default.
requestTimeoutCheckerInterval	1000	Camel 2.9.2: Configures how often Camel checks for request/reply over JMS. By default Camel checks every 1000ms. If a timeout occurs, then you can lower this interval by specifying the option requestTimeout.
subscriptionDurable	false	@deprecated: Enabled by default, if you specify durable subscription.
taskExecutor	null	Allows you to specify a custom task executor for the consumer.
taskExecutorsSpring2	null	Camel 2.6: To use when using Spring 2.x with consuming messages.
templateConnectionFactory	null	The JMS connection factory used for sending requests.
transactedInOut	false	@deprecated: Specifies whether to use transacted exchange pattern. Applies only to producer endpoint. See here for more details.
transactionManager	null	The Spring transaction manager to use.
transactionName	"JmsConsumer[destinationName]"	The name of the transaction to use.
transactionTimeout	null	The timeout value of the transaction (in seconds).
transferException	false	If enabled and you are using Request Reply message pattern, then the exception will be wrapped in a javax.jms.ObjectMessage. If the client is Camel, then Camel will catch the exception and return it to the producer. The caught exception is required to be serializable. If the exception is not serializable, then it will be wrapped in an outer exception such as org.apache.camel.CamelExecutionException and returned to the producer.
transferExchange	false	You can transfer the exchange over the wire in three ways: In body, Out body, Fault body, properties, exchange exception. This requires the payload to be serializable objects and log it at WARN level. You must enable this option on both the producer and consumer side if the payload is an Exchange and not a regular payload.
transferFault	false	Camel 2.17: If enabled and you are using Request Reply message pattern, then the exception will be wrapped in a SOAP fault (not exception) on the consumer side. The consumer can check for the key JmsConstants.JMS_TRANSFER_FAULT on the org.apache.camel.Message.setFault() method.
username	null	You may want to enable this when using Camel based such as CXF or spring-ws.
useMessageIDAsCorrelationID	false	The username for the connector factory.
useVersion102	false	Specifies whether JMSMessageID should always be used.

Message Mapping Between JMS and Camel

Camel automatically maps messages between `javax.jms.Message` and `org.apache.camel.Message`. When sending a JMS message, Camel converts the message body to the following JMS message types:

Body Type	JMS Message	Comment
<code>byte[]</code>	<code>javax.jms.BytesMessage</code>	
<code>java.io.File</code>	<code>javax.jms.BytesMessage</code>	
<code>java.io.InputStream</code>	<code>javax.jms.BytesMessage</code>	
<code>java.io.Reader</code>	<code>javax.jms.BytesMessage</code>	
<code>java.io.Serializable</code>	<code>javax.jms.ObjectMessage</code>	
<code>java.nio.ByteBuffer</code>	<code>javax.jms.BytesMessage</code>	
<code>Map</code>	<code>javax.jms.MapMessage</code>	
<code>org.w3c.dom.Node</code>	<code>javax.jms.TextMessage</code>	The DOM will be converted to string.

String	javax.jms.TextMessage
--------	-----------------------

When receiving a JMS message, Camel converts the JMS message to the following body type:
confluenceTableSmall

JMS Message	Body Type
javax.jms.BytesMessage	byte[]
javax.jms.MapMessage	Map<String, Object>
javax.jms.ObjectMessage	Object
javax.jms.TextMessage	String

Disabling Auto-Mapping of JMS Messages

You can use the `mapJmsMessage` option to disable the auto-mapping above. If disabled, Camel will not try to map the received JMS message, but instead uses it directly as the payload. This allows you to avoid the overhead of mapping and let Camel just pass through the JMS message. For instance, it even allows you to route `javax.jms.ObjectMessage` JMS messages with classes you do **not** have on the classpath.

Using a custom MessageConverter

You can use the `messageConverter` option to do the mapping yourself in a Spring `org.springframework.jms.support.converter.MessageConverter` class.

For example, in the route below we use a custom message converter when sending a message to the JMS order queue:
`javafrom("file://inbox/order") .to("jms:queue:order?messageConverter=#myMessageConverter");`

You can also use a custom message converter when consuming from a JMS destination.

Controlling the Mapping Strategy Selected

You can use the `jmsMessageType` option on the endpoint URL to force a specific message type for all messages. In the route below, we poll files from a folder and send them as `javax.jms.TextMessage` as we have forced the JMS producer endpoint to use text messages:

`javafrom("file://inbox/order") .to("jms:queue:order?jmsMessageType=Text");`

You can also specify the message type to use for each message by setting the header with the key `CamelJmsMessageType`. For example:

`javafrom("file://inbox/order") .setHeader("CamelJmsMessageType", JmsMessageType.Text) .to("jms:queue:order");`

The possible values are defined in the enum class `org.apache.camel.JmsMessageType`.

Message Format When Sending

An exchange sent via JMS must conform to the JMS Message spec. Camel therefore applies various translation and validation rules to both key names and key values of `exchange.in.headers`.

The following rules are applied to the `key names` of `exchange.in.headers`:

- Keys starting with `JMS` or `JMSX` are reserved.
- Key names must be literals or valid Java identifiers.
- Dot and hyphen characters are replaced (and the reverse when consuming) as follows:
 - The character `'.'` is replaced with the sequence `_DOT_`. The reverse replacement is applied when Camel consumes a message.
 - The character `'-'` is replaced with the sequence `_HYPHEN_`. The reverse replacement is applied when Camel consumes a message.
- The option `jmsKeyFormatStrategy` can be used to specify a custom key formatting strategy.

The following rules are applied to the `key values` of `exchange.in.headers`:

- Values must be either a primitive type or of its corresponding Java object type, e.g., `Integer`, `Long` or `Character`.
- The types `String`, `CharSequence`, `Date`, `BigDecimal` and `BigInteger` are all converted to their string representation.
- All other types will result in the key value being discarded.

If a header value is discarded Camel will log the incident using logging category

`org.apache.camel.component.jms.JmsBinding` at the `DEBUG` logging level. For example:

`text2008-07-09 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}`

Message Format When Receiving

Camel adds the following properties to the Exchange when it receives a message:
confluenceTableSmall

Property	Type	Description
----------	------	-------------

`org.apache.camel.jms.replydestination javax.jms.Destination` The reply destination.

Camel adds the following JMS properties to the In message headers when it receives a JMS message:
confluenceTableSmall

Header	Type	Description
JMSCorrelationID	String	The JMS correlation ID.
JMSDeliveryMode	int	The JMS delivery mode.
JMSDestination	javax.jms.Destination	The JMS destination.
JMSExpiration	long	The JMS expiration.
JMSMessageID	String	The JMS unique message ID.
JMSPriority	int	The JMS priority (with 0 as the lowest priority and 9 as the highest).
JMSRedelivered	boolean	Is the JMS message redelivered.
JMSReplyTo	javax.jms.Destination	The JMS reply-to destination.
JMSTimestamp	long	The JMS timestamp.
JMSType	String	The JMS type.
JMSXGroupID	string	The JMS group ID.

As all the above information is standard JMS you can check the JMS documentation for further details.

Using Camel to Send and Receive Messages Using `JMSReplyTo`

The JMS component is complex and you have to pay close attention to how it works in some cases. So this is a short summary of some of the areas/pitfalls to look for.

When Camel sends a message using its `JmsProducer` it checks the following conditions:

- The message Exchange Pattern (MEP)
- Whether a `JMSReplyTo` was set in the endpoint or in the message headers
- Whether any of the following options have been set on the JMS endpoint: `disableReplyTo`, `preserveMessageQos` or `explicitQosEnabled`.

All this can be a tad complex to understand and configure to support your use case.

JmsProducer

The `JmsProducer` behaves as follows, depending on configuration:
confluenceTableSmall

Exchange Pattern	Other options	Description
InOut	-	Camel will expect a reply, set a temporary <code>JMSReplyTo</code> , and after sending the message, it will start to listen for the reply message on the temporary queue.
InOut	<code>JMSReplyTo</code> is set	Camel will expect a reply and, after sending the message, it will start to listen for the reply message on the specified <code>JMSReplyTo</code> queue.
InOnly	-	Camel will send the message and not expect a reply.
InOnly	<code>JMSReplyTo</code> is set	<p>By default, Camel discards the <code>JMSReplyTo</code> destination and clears the <code>JMSReplyTo</code> header before sending the message. Camel then sends the message and does <i>not</i> expect a reply.</p> <p>Camel logs this in the log at <code>WARN</code> level (changed to <code>DEBUG</code> level from Camel 2.6 on). You can use <code>preserveMessageQoS=true</code> to instruct Camel to keep the <code>JMSReplyTo</code>.</p> <p>In all situations the <code>JmsProducer</code> does not expect any reply and thus continue after sending the message.</p>

JmsConsumer

The `JmsConsumer` behaves as follows, depending on configuration:
confluenceTableSmall

Exchange Pattern	Other options	Description
InOut	-	Camel will send the reply back to the <code>JMSReplyTo</code> queue.
InOnly	-	Camel will not send a reply back, as the pattern is <code>InOnly</code> .
-	<code>disablerReplyTo=true</code>	This option suppresses replies.

So pay attention to the message exchange pattern set on your exchanges.

If you send a message to a JMS destination in the middle of your route you can specify the exchange pattern to use, see more at Request Reply. This is useful if you want to send an `InOnly` message to a JMS topic:

```
javafrom("activemq:queue:in") .to("bean:validateOrder") .to(ExchangePattern.InOnly, "activemq:topic:order")
.to("bean:handleOrder");
```

Computing the Destination at Runtime

If you need to send messages to a lot of different JMS destinations, it makes sense to reuse a JMS endpoint and specify the real destination in a message header. This allows Camel to reuse the same endpoint, but send to different destinations. This greatly reduces the number of endpoints created and economizes on memory and thread resources.

You can specify the destination in the following headers:

confluenceTableSmall

Header	Type	Description
CamelJmsDestination	javax.jms.Destination	A destination object.
CamelJmsDestinationName	String	The destination name.

For example, the following route shows how you can compute a destination at run time and use it to override the destination appearing in the JMS URL:

```
javafrom("file://inbox") .to("bean:computeDestination") .to("activemq:queue:dummy");
```

The queue name, dummy, is just a placeholder. It must be provided as part of the JMS endpoint URL, but it will be ignored in this example.

In the `computeDestination` bean, specify the real destination by setting the `CamelJmsDestinationName` header as follows:

```
javapublic void setJmsHeader(Exchange exchange) { String id = .... exchange.getln().setHeader("CamelJmsDestinationName",
"order:" + id"); }
```

Then Camel will read this header and use it as the destination instead of the one configured on the endpoint. So, in this example Camel sends the message to `activemq:queue:order:2`, assuming the `id` value was 2.

If both the `CamelJmsDestination` and the `CamelJmsDestinationName` headers are set `CamelJmsDestination` will take priority. Note that the JMS producer removes both `CamelJmsDestination` and `CamelJmsDestinationName` headers from the exchange and does not propagate them to the created JMS message. This prevents accidental routing loops in scenarios where a message is forwarded to another JMS endpoint.

Configuring Different JMS Providers

A JMS provider can be configured in Spring XML as follows:{snippet:id=example|lang=xml|url=camel/trunk/components/camel-jms/src/test/resources/org/apache/camel/component/jms/jmsRouteUsingSpring.xml}

An unlimited number of JMS component instance can be created provided each has a *unique value for its id attribute*. The preceding example configures an activemq component. You could do the same to configure MQSeries, TibCo, BEA, Sonic etc.

Once named a JMS component can be referenced from an endpoint's URI. For example, given the component name activemq a URI can reference the component using the format activemq:[queue:]destinationName. The same approach applies to all JMS providers. This is achieved by the SpringCamelContext lazily fetching components from the spring context for the scheme name referenced in the Endpoint URLs then having the Component resolve the endpoint URI itself.

Using JNDI to Find the Connection Factory

If you are using a J2EE container, you might need to look up JNDI to find the JMS connectionFactory rather than use the usual `<bean>` mechanism in Spring. You can do this using Spring's factory bean or the new Spring XML namespace. For example:

```
xml<bean id="weblogic" class="org.apache.camel.component.jms.JmsComponent"> <property name="connectionFactory"
ref="myConnectionFactory"/> </bean> <jee:jndi-lookup id="myConnectionFactory" jndi-name="jms/connectionFactory"/>
```

See The jee schema in the Spring reference documentation for more details about JNDI lookup.

Concurrent Consuming

A common requirement with JMS is to consume messages concurrently in multiple threads in order to make an application more responsive. You can set the `concurrentConsumers` option to specify the number of threads servicing the JMS endpoint, as follows:

```
javafrom("jms:SomeQueue?concurrentConsumers=20") .bean(MyClass.class);
```

You can configure this option in one of the following ways:

- On the `JmsComponent`
- On the endpoint URI
- By invoking `setConcurrentConsumers()` directly on the `JmsEndpoint`.

Concurrent Consuming with `asyncConsumer`

Notice that each concurrent consumer will only pickup the next available message from the JMS broker, when the current message has been fully processed. You can set the option `asyncConsumer=true` to let the consumer pickup the next message from the JMS queue, while the previous message is being processed asynchronously (by the Asynchronous Routing Engine). See more details in the table on top of the page about the `asyncConsumer` option.

```
javafrom("jms:SomeQueue?concurrentConsumers=20&asyncConsumer=true") .bean(MyClass.class);
```

Request-Reply over JMS

Camel supports Request Reply over JMS. In essence the MEP of the Exchange should be `Inout` when you send a message to a JMS queue.

Camel offers a number of options to configure request/reply over JMS that influence performance and clustered environments. The table below summarizes the options.

confluenceTableSmall

Option	Performance	Cluster	Description
Temporary	Fast	Yes	A temporary queue is used as reply queue, and automatic created by Camel. To use this do not specify a <code>replyTo</code> queue name. And you can optionally configure <code>replyToType=Temporary</code> to make it stand out that temporary queues are in use.
Shared	Slow	Yes	A shared persistent queue is used as reply queue. The queue must be created beforehand, although some brokers can create them on the fly such as Apache ActiveMQ. To use this you must specify the <code>replyTo</code> queue name. And you can optionally configure <code>replyToType=Shared</code> to make it stand out that shared queues are in use. A shared queue can be used in a clustered environment with multiple nodes running this Camel application at the same time. All using the same shared reply queue. This is possible because JMS Message selectors are used to correlate expected reply messages; this impacts performance though. JMS Message selectors is slower, and therefore not as fast as <code>Temporary</code> or <code>Exclusive</code> queues. See further below how to tweak this for better performance.
Exclusive	Fast	No (*Yes)	An exclusive persistent queue is used as reply queue. The queue must be created beforehand, although some brokers can create them on the fly such as Apache ActiveMQ. To use this you must specify the <code>replyTo</code> queue name. And you must configure <code>replyToType=Exclusive</code> to instruct Camel to use exclusive queues, as shared is used by default, if a <code>replyTo</code> queue name was configured. When using exclusive reply queues, then JMS Message selectors are not in use, and therefore other applications must not use this queue as well. An exclusive queue cannot be used in a clustered environment with multiple nodes running this Camel application at the same time; as we do not have control if the reply queue comes back to the same node that sent the request message; that is why shared queues use JMS Message selectors to make sure of this. Though if you configure each Exclusive reply queue with an unique name per node, then you can run this in a clustered environment. As then the reply message will be sent back to that queue for the given node, that awaits the reply message.
concurrentConsumers	Fast	Yes	<p>Camel 2.10.3: Allows to process reply messages concurrently using concurrent message listeners in use. You can specify a range using the <code>concurrentConsumers</code> and <code>maxConcurrentConsumers</code> options.</p> <p>Note: That using shared reply queues may not work as well with concurrent listeners, so use this option with care.</p>
maxConcurrentConsumers	Fast	Yes	<p>Camel 2.10.3: Allows to process reply messages concurrently using concurrent message listeners in use. You can specify a range using the <code>concurrentConsumers</code> and <code>maxConcurrentConsumers</code> options.</p> <p>Note: That using shared reply queues may not work as well with concurrent listeners, so use this option with care.</p>

The `JmsProducer` detects the `Inout` and provides a `JMSReplyTo` header with the reply destination to be used. By default Camel uses a temporary queue, but you can use the `replyTo` option on the endpoint to specify a fixed reply queue (see more below about fixed reply queue).

Camel will automatically setup a consumer which listens on the reply queue, so you should **not** do anything. This consumer is a `SpringDefaultMessageListenerContainer` which listens for replies. However it's fixed to 1 concurrent consumer. That means replies will be processed in sequence as there are only 1 thread to process the replies. If you want to process replies faster, then we need to use concurrency. But **not** using the `concurrentConsumer` option. We should use the `threads` from the Camel DSL instead, as shown in the route below:

Instead of using threads, then use `concurrentConsumers` option if using Camel 2.10.3 or greater. See below for details.

```
javafrom(xxx).inOut().to("activemq:queue:foo").threads(5).to(yyy).to(zzz);
```

In this route we instruct Camel to route replies asynchronously using a thread pool with 5 threads.

From **Camel 2.10.3**: you can now configure the listener to use concurrent threads using the `concurrentConsumers` and `maxConcurrentConsumers` options. This allows you to easier configure this in Camel as shown below:

```
javafrom(xxx).InOut().to("activemq:queue:foo?concurrentConsumers=5").to(yyy).to(zzz);
```

Request-Reply over JMS Using a Shared Fixed Reply Queue

If you use a fixed reply queue when doing Request Reply over JMS as shown in the example below, then pay attention.

```
from(xxx).InOut().to("activemq:queue:foo?replyTo=bar").to(yyy);
```

In this example the fixed reply queue named "bar" is used. By default Camel assumes the queue is shared when using fixed reply queues, and therefore it uses a `JMSselector` to only pickup the expected reply messages (eg based on the `JMSCorrelationID`). See next section for exclusive fixed reply queues. That means its not as fast as temporary queues. You can speedup how often Camel will pull for reply messages using the `receiveTimeout` option. By default its 1000ms. So to make it faster you can set it to 250ms to pull 4 times per second as shown:

```
javafrom(xxx).InOut().to("activemq:queue:foo?replyTo=bar&receiveTimeout=250").to(yyy);
```

Notice this will cause the Camel to send pull requests to the message broker more frequent, and thus require more network traffic. It's generally recommended that temporary queues be used where possible.

Request-Reply over JMS Using an Exclusive Fixed Reply Queue

Available as of Camel 2.9

In the previous example, Camel would anticipate the fixed reply queue named bar was shared, and thus it uses a `JMSselector` to only consume reply messages which it expects. However there is a drawback doing this as JMS selectos is slower. Also the consumer on the reply queue is slower to update with new JMS selector ids. In fact it only updates when the `receiveTimeout` option times out, which by default is 1 second. So in theory the reply messages could take up till about 1 sec to be detected. On the other hand if the fixed reply queue is exclusive to the Camel reply consumer, then we can avoid using the JMS selectors, and thus be more performant. In fact as fast as using temporary queues. So in **Camel 2.9** onward we introduced the `ReplyToType` option which you can configure to `Exclusive` to tell Camel that the reply queue is exclusive as shown in the example below:

```
javafrom(xxx).InOut().to("activemq:queue:foo?replyTo=bar&replyToType=Exclusive").to(yyy);
```

Mind that the queue must be exclusive to each and every endpoint. So if you have two routes, then they each need an unique reply queue as shown in the next example:

```
javafrom(xxx).InOut().to("activemq:queue:foo?replyTo=bar&replyToType=Exclusive").to(yyy); from(aaa).  
InOut().to("activemq:queue:order?replyTo=order.reply&replyToType=Exclusive").to(bbb);
```

The same applies if you run in a clustered environment. Then each node in the cluster must use an unique reply queue name. As otherwise each node in the cluster may pickup messages which was intended as a reply on another node. For clustered environments its recommended to use shared reply queues instead.

Synchronizing Clocks Between Senders and Receivers

When doing messaging between systems, its desirable that the systems have synchronized clocks. For example when sending a JMS message, then you can set a time to live value on the message. Then the receiver can inspect this value, and determine if the message is already expired, and thus drop the message instead of consume and process it. However this requires that both sender and receiver have synchronized clocks. If you are using ActiveMQ then you can use the timestamp plugin to synchronize clocks.

About Time To Live

Read first above about synchronized clocks.

When you do request/reply, `Inout`, over JMS Camel uses a timeout on the sender side, which is default 20 seconds, taken from the `requestTimeout` option. You can control this by setting a higher/lower value. However, the time to live value is still set on the JMS message being sent. This therefore requires that system clocks be synchronized between the systems. If they are not, then you may want to disable the time to live value being set. This is now possible using the `disableTimeToLive` option from **Camel 2.8** onward. So if you set this option to `disableTimeToLive=true`, then Camel does **not** set any time to live value when sending JMS messages. **But** the request timeout is still active. So for example if you do request/reply over JMS and have disabled time to live, then Camel will still use a timeout by 20 seconds (the `requestTimeout` option). That option can of course also be configured. So the two options `requestTimeout` and `disableTimeToLive` gives you fine grained control when doing request/reply.

From **Camel 2.13/2.12.3**: you can provide a header in the message to override and use as the request timeout value instead of the endpoint configured value. For example:

```
javafrom("direct:someWhere").to("jms:queue:foo?replyTo=bar&requestTimeout=30s").to("bean:processReply");
```

In the route above we have a endpoint configured `requestTimeout` of 30 seconds. So Camel will wait up till 30 seconds for that reply message to come back on the bar queue. If no reply message is received then a

`org.apache.camel.ExchangeTimedoutException` is set on the Exchange and Camel continues routing the message, which would then fail due to the exception, and Camel's error handler reacts.

If you want to use a per message timeout value, you can set the header with key

`org.apache.camel.component.jms.JmsConstants#JMS_REQUEST_TIMEOUT` which has constant value

`CamelJmsRequestTimeout` with a timeout value as long type.

For example we can use a bean to compute the timeout value per individual message, such as calling the "whatIsTheTimeout" method on the service bean as shown below:

```
javafrom("direct:someWhere") .setHeader("CamelJmsRequestTimeout", method(ServiceBean.class, "whatIsTheTimeout"))
.to("jms:queue:foo?replyTo=bar&requestTimeout=30s") .to("bean:processReply");
When you do fire and forget (inout) over JMS Camel will, by default, not set a time to live value on the message. The value can be configured using the timeToLive option. For example to indicate a 5 sec., you set timeToLive=5000. The option disableTimeToLive can be used to force disabling the time to live, also for inonly messaging. The requestTimeout option is not being used for inonly messaging.
```

Enabling Transacted Consumption

`transactedConsumption`

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

- `transacted = true`
- `transactionManager = <SomeTransactionManager>` (typically the `JmsTransactionManager`)

See the Transactional Client EIP pattern for further details.

Transactions and [Request Reply] over JMS

When using Request Reply over JMS you cannot use a single transaction; JMS will not send any messages until a commit is performed, so the server side won't receive anything at all until the transaction commits. Therefore to use Request Reply you must commit a transaction after sending the request and then use a separate transaction for receiving the response.

To address this issue the JMS component uses different properties to specify transaction use for oneway messaging and request reply messaging:

- The `transacted` property applies **only** to the `inonly` message Exchange Pattern (MEP).
- The `transactedInOut` property applies to the `inout` (Request Reply) message Exchange Pattern (MEP). If you want to use transactions with the `inout` (Request Reply) message Exchange Pattern (MEP), you *must* set `transactedInOut=true`.

Available as of Camel 2.10

You can leverage the DMLC transacted session API using the following properties on component/endpoint:

- `transacted = true`
- `lazyCreateTransactionManager = false`

The benefit of doing so is that the `cacheLevel` setting will be honored when using local transactions without a configured `TransactionManager`. When a `TransactionManager` is configured, no caching happens at DMLC level and its necessary to rely on a pooled connection factory. For more details about this kind of setup see [here](#) and [here](#).

Using `JMSReplyTo` For Late Replies

When using Camel as a JMS listener, it sets an Exchange property with the value of the `ReplyTo` `javax.jms.Destination` object, having the key `ReplyTo`. You can obtain this `Destination` as follows:

```
javaDestination replyDestination = exchange.getln().getHeader(JmsConstants.JMS_REPLY_DESTINATION, Destination.class);
And then later use it to send a reply using regular JMS or Camel.
```

```
java// We need to pass in the JMS component (this example uses ActiveMQ): JmsEndpoint endpoint =
JmsEndpoint.newInstance(replyDestination, activeMQComponent); // Now that we have the endpoint we can use regular Camel
API to send a message to it template.sendBody(endpoint, "Here is the late reply.");
```

A different solution to sending a reply is to provide the `replyToDestination` object in the same Exchange property when sending. Camel will then pick up this property and use it for the real destination. The endpoint URI must include a dummy destination, however.

Example:

```
java// we pretend to send it to some non existing dummy queue template.send("activemq:queue:dummy, new Processor() { public
void process(Exchange exchange) throws Exception { // and here we override the destination with the ReplyTo destination object
so the message is sent to there instead of dummy exchange.getln().setHeader(JmsConstants.JMS_DESTINATION,
replyDestination); exchange.getln().setBody("Here is the late reply."); } }
```

Using a Request Timeout

In the sample below we send a Request Reply style message Exchange (we use the `requestBody` method = `inout`) to the slow queue for further processing in Camel and we wait for a return reply:([snippet:id=e1|lang=java|url=camel/trunk/components/camel-jms/src/test/java/org/apache/camel/component/jms/JmsRouteTimeoutTest.java](#))

Examples

JMS is used in many examples for other components as well. But we provide a few samples below to get started.

Receiving from JMS

In the following sample we configure a route that receives JMS messages and routes the message to a POJO:

```
java from("jms:queue:foo") .to("bean:myBusinessLogic");
```

You can of course use any of the EIP patterns so the route can be context based. For example, here's how to filter an order topic for the big spenders:

```
javafrom("jms:topic:OrdersTopic") .filter().method("myBean", "isGoldCustomer") .to("jms:queue:BigSpendersQueue");
```

Sending to JMS

In the sample below we poll a file folder and send the file content to a JMS topic. As we want the content of the file as a `TextMessage` instead of a `BytesMessage`, we need to convert the body to a `String`:

```
javafrom("file://orders") .convertBodyTo(String.class) .to("jms:topic:OrdersTopic");
```

Using Annotations

Camel also has annotations so you can use POJO Consuming and POJO Producing.

Spring DSL Example

The preceding examples use the Java DSL. Camel also supports Spring XML DSL. Here is the big spender sample using Spring DSL:

```
xml<route> <from uri="jms:topic:OrdersTopic"/> <filter> <method bean="myBean" method="isGoldCustomer"/> <to uri="jms:queue:BigSpendersQueue"/> </filter> </route>
```

Other Examples

JMS appears in many of the examples for other components and EIP patterns, as well in this Camel documentation. So feel free to browse the documentation. If you have time, check out the this tutorial that uses JMS but focuses on how well Spring Remoting and Camel works together Tutorial-JmsRemoting.

Using JMS as a Dead Letter Queue Storing Exchange

Normally, when using JMS as the transport, it only transfers the body and headers as the payload. If you want to use JMS with a Dead Letter Channel, using a JMS queue as the Dead Letter Queue, then normally the caused Exception is not stored in the JMS message. You can, however, use the `transferExchange` option on the JMS dead letter queue to instruct Camel to store the entire Exchange in the queue as a `javax.jms.ObjectMessage` that holds a `org.apache.camel.impl.DefaultExchangeHolder`. This allows you to consume from the Dead Letter Queue and retrieve the caused exception from the Exchange property with the key Exchange. `EXCEPTION_CAUGHT`.

Example:
 java// setup error handler to use JMS as queue and store the entire Exchange
 errorHandler(deadLetterChannel("jms:queue:dead?transferExchange=true"));
 Then you can consume from the JMS queue and analyze the problem:
 javafrom("jms:queue:dead") .to("bean:myErrorHandler"); // and in our bean String body = exchange.getln().getBody(); Exception cause = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.class); // The cause message is String problem = cause.getMessage();

Using JMS as a Dead Letter Channel for Storing Error Only

You can use JMS to store the cause error message or to store a custom body, which you can initialize yourself. The following example uses the Message Translator EIP to do a transformation on the failed exchange before it is moved to the JMS dead letter queue:

```
java// We sent it to a seda dead queue first errorHandler(deadLetterChannel("seda:dead")); // On the seda dead queue we can do the custom transformation before its sent to the JMS queue from("seda:dead") .transform(exceptionMessage()) .to("jms:queue:dead");
```

Here we only store the original cause error message in the transform. You can, however, use any Expression to send whatever you like. For example, you can invoke a method on a Bean or use a custom processor.

Sending an Inonly Message and Keeping the JMSReplyTo Header

When sending to a JMS destination using `camel-jms` the producer will use the MEP to detect if it's `Inonly` or `Inout` messaging. However, there can be times where you want to send an `Inonly` message but keeping the `JMSReplyTo` header. To do so you have to instruct Camel to keep it, otherwise the `JMSReplyTo` header will be dropped.

For example to send an `Inonly` message to the `foo` queue, but with a `JMSReplyTo` with `bar` queue you can do as follows:

```
javatemplate.send("activemq:queue:foo?preserveMessageQos=true", new Processor() { public void process(Exchange exchange) throws Exception { exchange.getln().setBody("World"); exchange.getln().setHeader("JMSReplyTo", "bar"); } });
```

Note: we use `preserveMessageQos=true` to instruct Camel to keep the `JMSReplyTo` header.

Setting JMS Provider Options on the Destination

Some JMS providers, like IBM's WebSphere MQ need options to be set on the JMS destination. For example, you may need to specify the targetClient option. Since targetClient is a WebSphere MQ option and not a Camel URI option, you need to set that on the JMS destination name like so:

```
java... .setHeader("CamelJmsDestinationName", constant("queue:///MY_QUEUE?targetClient=1")) .to("wmq:queue:MY_QUEUE?useMessageIDAsCorrelationID=true");
```

Some versions of Websphere MQ do not accept this option on the destination name. The following exception is raised when this happens:

```
com.ibm.msg.client.jms.DetailedJMSEException: JMSCC0005: The specified value 'MY_QUEUE?targetClient=1' is not allowed for 'XMSC_DESTINATION_NAME'
```

A workaround is to use a custom DestinationResolver:

```
javaJmsComponent wmq = new JmsComponent(connectionFactory); wmq.setDestinationResolver(new DestinationResolver(){ public Destination resolveDestinationName(Session session, String destinationName, boolean pubSubDomain) throws JMSEException { MQQueueSession wmqSession = (MQQueueSession) session; return wmqSession.createQueue("queue://" + destinationName + "?targetClient=1"); } });
```

Endpoint See Also

- Transactional Client
- Bean Integration
- Tutorial-JmsRemoting
- JMSTemplate gotchas

JMX COMPONENT

Available as of Camel 2.6

Standard JMX Consumer Configuration

Component allows consumers to subscribe to an mbean's Notifications. The component supports passing the Notification object directly through the Exchange or serializing it to XML according to the schema provided within this project. This is a consumer only component. Exceptions are thrown if you attempt to create a producer for it.

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

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

URI Format

The component can connect to the local platform mbean server with the following URI:

jmx://platform?options

A remote mbean server url can be provided following the initial JMX scheme like so:

jmx:service:jmx:rmi://localhost:1099/jmxrmi?options

You can append query options to the URI in the following format, ?options=value&option2=value&...

URI Options

confluenceTableSmall

Property	Required	Default	Description
format		xml	Format for the message body. Either "xml" or "raw". If xml, the notification is serialized to xml. If raw, then the raw java object is set as the body.
user			Credentials for making a remote connection.
password			Credentials for making a remote connection.
objectDomain	yes		The domain for the mbean you're connecting to.
objectName			The name key for the mbean you're connecting to. This value is mutually exclusive with the object properties that get passed. (see below)
notificationFilter			Reference to a bean that implements the <code>NotificationFilter</code> . The <code>#ref</code> syntax should be used to reference the bean via the Registry.
handback			Value to handback to the listener when a notification is received. This value will be put in the message header with the key "jmx.handback"
testConnectionOnStartup	true		Camel 2.11 If true, the consumer will throw an exception when

		unable to establish the JMX connection upon startup. If false, the consumer will attempt to establish the JMX connection every 'x' seconds until the connection is made – where 'x' is the configured <i>reconnectDelay</i> .
reconnectOnConnectionFailure	false	Camel 2.11 If true, the consumer will attempt to reconnect to the JMX server when any connection failure occurs. The consumer will attempt to re-establish the JMX connection every 'x' seconds until the connection is made– where 'x' is the configured <i>reconnectDelay</i> .
reconnectDelay	10 seconds	Camel 2.11 The number of seconds to wait before retrying creation of the initial connection or before reconnecting a lost connection.

ObjectName Construction

The URL must always have the objectDomain property. In addition, the URI must contain either objectName or one or more properties that start with "key."

Domain with Name property

When the objectName property is provided, the following constructor is used to build the ObjectName? for the mbean:
ObjectName(String domain, String key, String value)

The key value in the above will be "name" and the value will be the value of the objectName property.

Domain with Hashtable

ObjectName(String domain, Hashtable<String, String> table)

The Hashtable is constructed by extracting properties that start with "key." The properties will have the "key." prefixed stripped prior to building the Hashtable. This allows the URI to contain a variable number of properties to identify the mbean.

Example

```
{snippet:id=e1|lang=java|url=camel/trunk/examples/camel-example-jmx/src/main/java/org/apache/camel/example/jmx/MyRouteBuilder.java}
Full example
```

Monitor Type Consumer

Available as of Camel 2.8

One popular use case for JMX is creating a monitor bean to monitor an attribute on a deployed bean. This requires writing a few lines of Java code to create the JMX monitor and deploy it. As shown below:

```
java CounterMonitor monitor = new CounterMonitor(); monitor.addObservedObject(makeObjectName("simpleBean"));
monitor.setObservedAttribute("MonitorNumber"); monitor.setNotify(true); monitor.setInitThreshold(1);
monitor.setGranularityPeriod(500); registerBean(monitor, makeObjectName("counter")); monitor.start();
```

The 2.8 version introduces a new type of consumer that automatically creates and registers a monitor bean for the specified objectName and attribute. Additional endpoint attributes allow the user to specify the attribute to monitor, type of monitor to create, and any other required properties. The code snippet above is condensed into a set of endpoint properties. The consumer uses these properties to create the CounterMonitor, register it, and then subscribe to its changes. All of the JMX monitor types are supported.

Example

```
java from("jmx:platform?objectDomain=myDomain&objectName=simpleBean&" +
"monitorType=counter&observedAttribute=MonitorNumber&initThreshold=1&" + "granularityPeriod=500").to("mock:sink");
The example above will cause a new Monitor Bean to be created and deployed to the local mbean server that monitors the "MonitorNumber" attribute on the "simpleBean." Additional types of monitor beans and options are detailed below. The newly deployed monitor bean is automatically undeployed when the consumer is stopped.
```

URI Options for Monitor Type

property	type	applies to	description
monitorType	enum	all	one of counter, guage, string
observedAttribute	string	all	the attribute being observed
granularityPeriod	long	all	granularity period (in millis) for the attribute being observed. As per JMX, default is 10 seconds

initThreshold	number	counter	initial threshold value
offset	number	counter	offset value
modulus	number	counter	modulus value
differenceMode	boolean	counter, gauge	true if difference should be reported, false for actual value
notifyHigh	boolean	gauge	high notification on/off switch
notifyLow	boolean	gauge	low notification on/off switch
highThreshold	number	gauge	threshold for reporting high notification
lowThreshold	number	gauge	threshold for reporting low notification
notifyDiffer	boolean	string	true to fire notification when string differs
notifyMatch	boolean	string	true to fire notification when string matches
stringToCompare	string	string	string to compare against the attribute value

The monitor style consumer is only supported for the local mbean server. JMX does not currently support remote deployment of mbeans without either having the classes already remotely deployed or an adapter library on both the client and server to facilitate a proxy deployment.

Endpoint See Also

- Camel JMX

JPA COMPONENT

The **jpa** component enables you to store and retrieve Java objects from persistent storage using EJB 3's Java Persistence Architecture (JPA), which is a standard interface layer that wraps Object/Relational Mapping (ORM) products such as OpenJPA, Hibernate, TopLink, and so on.

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

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

Sending to the endpoint

You can store a Java entity bean in a database by sending it to a JPA producer endpoint. The body of the *In* message is assumed to be an entity bean (that is, a POJO with an @Entity annotation on it) or a collection or array of entity beans.

If the body is a List of entities, make sure to use **entityType=java.util.ArrayList** as a configuration passed to the producer endpoint.

If the body does not contain one of the previous listed types, put a Message Translator in front of the endpoint to perform the necessary conversion first.

From **Camel 2.19** onwards you can use **query**, **namedQuery** and **nativeQuery** option for the producer as well to retrieve a set of entities or execute bulk update/delete.

Consuming from the endpoint

Consuming messages from a JPA consumer endpoint removes (or updates) entity beans in the database. This allows you to use a database table as a logical queue: consumers take messages from the queue and then delete/update them to logically remove them from the queue.

If you do not wish to delete the entity bean when it has been processed (and when routing is done), 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 when it has been processed (and when routing is done).

From **Camel 2.13** onwards you can use **@PreConsumed** which will be invoked on your entity bean before it has been processed (before routing).

If you are consuming a lot (100K+) of rows and experience OutOfMemory problems you should set the maximumResults to sensible value.

Note: Since **Camel 2.18**, JPA now includes a **JpaPollingConsumer** implementation that better supports Content Enricher using **pollEnrich()** to do an on-demand poll that returns either none, one or a list of entities as the result.

URI format

jpa:*entityClassName*[?options]

For sending to the endpoint, the *entityClassName* is optional. If specified, it helps the Type Converter to ensure the body is of the correct type.

For consuming, the *entityClassName* is mandatory.

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

Options

Name	Default Value	Description
entityType	<code>entityClassName</code>	Overrides the <code>entityClassName</code> from the URI.
persistenceunit	camel	The JPA persistence unit used by default.
consumeDelete	true	JPA consumer only: If true, the entity is deleted after it is consumed; if false, the entity is not deleted.
consumeLockEntity	true	JPA consumer only: Specifies whether or not to set an exclusive lock on each entity bean while processing the results from polling.
flushOnSend	true	JPA producer only: Flushes the EntityManager after the entity bean has been persisted.
maximumResults	-1	JPA consumer only: Set the maximum number of results to retrieve on the Query. Camel 2.19: it's also used for the producer when it executes a query.
transactionManager	null	This option is Registry based which requires the # notation so that the given <code>transactionManager</code> being specified can be looked up properly, e.g. <code>transactionManager=#myTransactionManager</code> . It specifies the transaction manager to use. If none provided, Camel will use a <code>JpaTransactionManager</code> by default. Can be used to set a JTA transaction manager (for integration with an EJB container).
consumer.delay	500	JPA consumer only: Delay in milliseconds between each poll.
consumer.initialDelay	1000	JPA consumer only: Milliseconds before polling starts.
consumer.useFixedDelay	false	JPA consumer only: Set to true to use fixed delay between polls, otherwise fixed rate is used. See <code>ScheduledExecutorService</code> in JDK for details.
maxMessagesPerPoll	0	JPA consumer only: An integer value to define the maximum number of messages to gather per poll. By default, no maximum is set. Can be used to avoid polling many thousands of messages when starting up the server. Set a value of 0 or negative to disable.
consumer.query		JPA consumer only: To use a custom query when consuming data.
consumer.namedQuery		JPA consumer only: To use a named query when consuming data.
consumer.nativeQuery		JPA consumer only: To use a custom native query when consuming data. You may want to use the option <code>consumer.resultClass</code> also when using native queries.
consumer.parameters		Camel 2.12: JPA consumer only: This option is Registry based which requires the # notation. This key/value mapping is used for building the query parameters. It's expected to be of the generic type <code>java.util.Map<String, Object></code> where the keys are the named parameters of a given JPA query and the values are their corresponding effective values you want to select for.
consumer.resultClass		Camel 2.7: JPA consumer only: Defines the type of the returned payload (we will call <code>entityManager.createNativeQuery(nativeQuery, resultClass)</code> instead of <code>entityManager.createNativeQuery(nativequery)</code>). Without this option, we will return an object array. Only has an affect when using in conjunction with native query when consuming data.
consumer.transacted	false	Camel 2.7.5/2.8.3/2.9: JPA consumer only: Whether to run the consumer in transacted mode, by which all messages will either commit or rollback, when the entire batch has been processed. The default behavior (false) is to commit all the previously successfully processed messages, and only rollback the last failed message.
consumer.lockModeType	WRITE	Camel 2.11.2/2.12: To configure the lock mode on the consumer. The possible values is defined in the enum <code>javax.persistence.LockModeType</code> . The default value is changed to <code>PESSIMISTIC_WRITE</code> since Camel 2.13 .
consumer.skipLockedEntity	false	Camel 2.13: To configure whether to use NOWAIT on lock and silently skip the entity.
usePersist	false	Camel 2.5: JPA producer only: Indicates to use <code>entityManager.persist(entity)</code> instead of <code>entityManager.merge(entity)</code> . Note:

		entityManager.persist(entity) doesn't work for detached entities (where the EntityManager has to execute an UPDATE instead of an INSERT query)!
joinTransaction	true	Camel 2.12.3: camel-jpa will join transaction by default from Camel 2.12 onwards. You can use this option to turn this off, for example if you use LOCAL_RESOURCE and join transaction doesn't work with your JPA provider. This option can also be set globally on the JpaComponent, instead of having to set it on all endpoints.
usePassedInEntityManager	false	Camel 2.12.4/2.13.1 JPA producer only: If set to true, then Camel will use the EntityManager from the header JpaConstants.ENTITYMANAGER instead of the configured entity manager on the component/endpoint. This allows end users to control which entity manager will be in use.
sharedEntityManager	false	Camel 2.16: whether to use spring's SharedEntityManager for the consumer/producer. A good idea may be to set joinTransaction=false if this option is true, as sharing the entity manager and mixing transactions is not a good idea.
query		To use a custom query. Camel 2.19: it can be used for producer as well.
namedQuery		To use a named query. Camel 2.19: it can be used for producer as well.
nativeQuery		To use a custom native query. You may want to use the option resultClass also when using native queries. Camel 2.19: it can be used for producer as well.
parameters		This option is Registry based which requires the # notation. This key/value mapping is used for building the query parameters. It is expected to be of the generic type java.util.Map<String, Object> where the keys are the named parameters of a given JPA query and the values are their corresponding effective values you want to select for. Camel 2.19: it can be used for producer as well. When it's used for producer, Simple expression can be used as a parameter value. It allows you to retrieve parameter values from the message body header and etc.
resultClass		Defines the type of the returned payload (we will call entityManager.createNativeQuery(nativeQuery, resultClass) instead of entityManager.createNativeQuery(nativeQuery)). Without this option, we will return an object array. Only has an affect when using in conjunction with native query. Camel 2.19: it can be used for producer as well.
useExecuteUpdate		Camel 2.19: JPA producer only: To configure whether to use executeUpdate() when producer executes a query. When you use INSERT, UPDATE or DELETE statement as a named query, you need to specify this option to 'true'.

Message Headers

Camel adds the following message headers to the exchange:
confluenceTableSmall

Header	Type	Description
CamelJpaTemplate	JpaTemplate	Not supported anymore since Camel 2.12: The JpaTemplate object that is used to access the entity bean. You need this object in some situations, for instance in a type converter or when you are doing some custom processing. See CAMEL-5932 for the reason why the support for this header has been dropped.
CamelEntityManager	EntityManager	Camel 2.12: JPA consumer / Camel 2.12.2: JPA producer: The JPA EntityManager object being used by JpaConsumer or JpaProducer.

Configuring EntityManagerFactory

It's strongly advised to configure the JPA component to use a specific EntityManagerFactory instance. If failed to do so each JpaEndpoint will auto create their own instance of EntityManagerFactory which most often is not what you want.

For example, you can instantiate a JPA component that references the myEMFactory entity manager factory, as follows:

```
xml<bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent"> <property name="entityManagerFactory" ref="myEMFactory"/> </bean>
```

In **Camel 2.3** the JpaComponent will auto lookup the EntityManagerFactory from the Registry which means you do not need to configure this on the JpaComponent as shown above. You only need to do so if there is ambiguity, in which case Camel will log a WARN.

Configuring TransactionManager

Since **Camel 2.3** the `JpaComponent` will auto lookup the `TransactionManager` from the Registry. If Camel won't find any `TransactionManager` instance registered, it will also look up for the `TransactionTemplate` and try to extract `TransactionManager` from it.

If none `TransactionTemplate` is available in the registry, `JpaEndpoint` will auto create their own instance of `TransactionManager` which most often is not what you want.

If more than single instance of the `TransactionManager` is found, Camel will log a WARN. In such cases you might want to instantiate and explicitly configure a JPA component that references the `myTransactionManager` transaction manager, as follows:

```
xml<bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent"> <property name="entityManagerFactory" ref="myEMFactory"/> <property name="transactionManager" ref="myTransactionManager"/> </bean>
```

Using a consumer with a named query

For consuming only selected entities, you can use the `consumer.namedQuery` URI query option. First, you have to define the named query in the JPA Entity class:

```
@Entity @NamedQuery(name = "step1", query = "select x from MultiSteps x where x.step = 1") public class MultiSteps { ... }
After that you can define a consumer uri like this one:  
from("jpa://org.apache.camel.examples.MultiSteps?consumer.namedQuery=step1") .to("bean:myBusinessLogic");
```

Using a consumer with a query

For consuming only selected entities, you can use the `consumer.query` URI query option. You only have to define the query option:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.query=select o from org.apache.camel.examples.MultiSteps o where o.step = 1") .to("bean:myBusinessLogic");
```

Using a consumer with a native query

For consuming only selected entities, you can use the `consumer.nativeQuery` URI query option. You only have to define the native query option:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.nativeQuery=select * from MultiSteps where step = 1") .to("bean:myBusinessLogic");
```

If you use the native query option, you will receive an object array in the message body.

Using a producer with a named query

For retrieving selected entities or execute bulk update/delete, you can use the `namedQuery` URI query option. First, you have to define the named query in the JPA Entity class:

```
@Entity @NamedQuery(name = "step1", query = "select x from MultiSteps x where x.step = 1") public class MultiSteps { ... }
After that you can define a producer uri like this one:  
from("direct:namedQuery") .to("jpa://org.apache.camel.examples.MultiSteps?namedQuery=step1");
```

Using a producer with a query

For retrieving selected entities or execute bulk update/delete, you can use the `query` URI query option. You only have to define the query option:

```
from("direct:query") .to("jpa://org.apache.camel.examples.MultiSteps?query=select o from org.apache.camel.examples.MultiSteps o where o.step = 1");
```

Using a producer with a native query

For retrieving selected entities or execute bulk update/delete, you can use the `nativeQuery` URI query option. You only have to define the native query option:

```
from("direct:nativeQuery") .to("jpa://org.apache.camel.examples.MultiSteps?  
resultClass=org.apache.camel.examples.MultiSteps&nativeQuery=select * from MultiSteps where step = 1");
If you use the native query option without specifying resultClass, you will receive an object array in the message body.
```

Example

See Tracer Example for an example using JPA to store traced messages into a database.

Using the JPA based idempotent repository

In this section we will use the JPA based idempotent repository.

First we need to setup a `persistence-unit` in the `persistence.xml` file:
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-jpa/src/test/resources/META-INF/persistence.xml}Second we have to

```
setup a org.springframework.orm.jpa.JpaTemplate which is used by the
org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository:
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-
jpa/src/test/resources/org/apache/camel/processor/jpa/spring.xml}Afterwards we can configure our
org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository:
{snippet:id=jpaStore|lang=xml|url=camel/trunk/components/camel-
jpa/src/test/resources/org/apache/camel/processor/jpa/fileConsumer.JpaldempotentTest-config.xml}And finally we can create our
JPA idempotent repository in the spring XML file as well:
xml<camelContext xmlns="http://camel.apache.org/schema/spring"> <route id="JpaMessageIdRepositoryTest"> <from
uri="direct:start" /> <idempotentConsumer messageIdRepositoryRef="jpaStore" /> <header>messageId</header> <to
uri="mock:result" /> </idempotentConsumer> </route> </camelContext> When running this Camel component tests inside your
IDE
```

In case you run the tests of this component directly inside your IDE (and not necessarily through Maven itself) then you could spot exceptions like:

```
java.org.springframework.transaction.CannotCreateTransactionException: Could not open JPA EntityManager for transaction;
nested exception is <openjpa-2.2.1-r422266:1396819 nonfatal user error> org.apache.openjpa.persistence.ArgumentException:
This configuration disallows runtime optimization, but the following listed types were not enhanced at build time or at class load
time with a javaagent: "org.apache.camel.examples.SendEmail". at
org.springframework.orm.jpa.JpaTransactionManager.doBegin(JpaTransactionManager.java:427) at
org.springframework.transaction.support.AbstractPlatformTransactionManager.getTransaction(AbstractPlatformTransactionManager
at org.springframework.transaction.support.TransactionTemplate.execute(TransactionTemplate.java:127) at
org.apache.camel.processor.jpa.JpaRouteTest.cleanupRepository(JpaRouteTest.java:96) at
org.apache.camel.processor.jpa.JpaRouteTest.createCamelContext(JpaRouteTest.java:67) at
org.apache.camel.test.junit4.CamelTestSupport.setUp(CamelTestSupport.java:238) at
org.apache.camel.test.junit4.CamelTestSupport.setUp(CamelTestSupport.java:208)
The problem here is that the source has been compiled/recompiled through your IDE and not through Maven itself which would
enhance the byte-code at build time. To overcome this you would need to enable dynamic byte-code enhancement of OpenJPA.
As an example assuming the current OpenJPA version being used in Camel itself is 2.2.1, then as running the tests inside your
favorite IDE you would need to pass the following argument to the JVM:
-javaagent:<path_to_your_local_m2_cache>/org/apache/openjpa/openjpa/2.2.1/openjpa-2.2.1.jar
```

Then it will all become green again 😊

Endpoint See Also

- Tracer Example

JT/400 COMPONENT

The jt400 component allows you to exchanges messages with an AS/400 system using data queues.

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

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jt400</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI format

```
jt400://user:password@system/QSYS.LIB/LIBRARY.LIB/QUEUE.DTAQ[?options]
```

To call remote program (Camel 2.7)

```
jt400://user:password@system/QSYS.LIB/LIBRARY.LIB/program.PGM[?options]
```

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

URI options

For the data queue message exchange:

Name	Default value	Description
ccsid	default system CCSID	Specifies the CCSID to use for the connection with the AS/400 system.
format	text	Specifies the data format for sending messages. Valid options are: text (represented by String) and binary (represented by byte[])
consumer.delay	500	Delay (milliseconds) between each poll
consumer.initialdelay	1000	
consumer.fixeddelay	false	true to use fixed delay between polls, otherwise use random delay. See SchedulingExecutorService in JDK for details.
guiaavailable	false	Camel 2.8: Specifies whether AS400 prompting is enabled in the environment running Camel.
keyed	false	Camel 2.10: Whether to use keyed or non-keyed data queues.
searchkey	null	Camel 2.10: Search key for keyed data queues.
searchtype	EQ	Camel 2.10: Search type which can be a value of EQ, NE, LT, LE, GT, or GE.

connectionPool	AS400ConnectionPool instance	Camel 2.10: Reference to an com.ibm.as400.access AS400ConnectionPool instance in the Registry. This is used for obtaining connections to the AS/400 system. The look up notation ("# character) should be used.
Secured	false	Camel 2.16: Whether to use SSL connections to the AS400
For the remote program call (Camel 2.7)		
Name	Default value	Description
outputFieldsIdx		Specifies which fields (program parameters) are output parameters.
fieldsLength		Specifies the fields (program parameters) length as in the AS/400 program definition.
format	text	Camel 2.10: Specifies the data format for reading messages. Valid options are: <code>text</code> (represented by <code>String</code>) and <code>binary</code> (represented by <code>byte[]</code>)
guiAvailable	false	Camel 2.8: Specifies whether AS/400 prompting is enabled in the environment running Camel.
connectionPool	AS400ConnectionPool instance	Camel 2.10: Reference to an com.ibm.as400.access AS400ConnectionPool instance in the Registry. This is used for obtaining connections to the AS/400 system. The look up notation ("# character) should be used.

Usage

When configured as a consumer endpoint, the endpoint will poll a data queue on a remote system. For every entry on the data queue, a new Exchange is sent with the entry's data in the `In` message's body, formatted either as a `String` or a `byte[]`, depending on the format. For a provider endpoint, the `In` message body contents will be put on the data queue as either raw bytes or text.

Connection pool

Available as of Camel 2.10

Connection pooling is in use from Camel 2.10 onwards. You can explicitly configure a connection pool on the `Jt400Component`, or as an uri option on the endpoint.

Remote program call (Camel 2.7)

This endpoint expects the input to be either a `String` array or `byte[]` array (depending on format) and handles all the CCSID handling through the native jt400 library mechanisms. A parameter can be *omitted* by passing null as the value in its position (the remote program has to support it). After the program execution the endpoint returns either a `String` array or `byte[]` array with the values as they were returned by the program (the input only parameters will contain the same data as the beginning of the invocation)

This endpoint does not implement a provider endpoint!

Example

In the snippet below, the data for an exchange sent to the `direct:george` endpoint will be put in the data queue `PENNYLANE` in library `BEATLES` on a system named `LIVERPOOL`.

Another user connects to the same data queue to receive the information from the data queue and forward it to the `mock:ringo` endpoint.

```
public class Jt400RouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("direct:george").to("jt400://GEORGE:EGROEG@LIVERPOOL/QSYS.LIB/BEATLES.LIB/PENNYLANE.DTAQ");
        from("jt400://RINGO:OGNIR@LIVERPOOL/QSYS.LIB/BEATLES.LIB/PENNYLANE.DTAQ").to("mock:ringo");
    }
}
```

Remote program call example (Camel 2.7)

In the snippet below, the data Exchange sent to the `direct:work` endpoint will contain three string arguments for the "compute" program in the library "assets". This program will write the output values in the 2nd and 3rd parameters. All the parameters will be sent to the `direct:play` endpoint.

```
public class Jt400RouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("direct:work").to("jt400://GRUPO:ATWORK@server/QSYS.LIB/assets.LIB/compute.PGM?fieldsLength=10,10,512&outputFieldsIdx=2,3").to("direct:play");
    }
}
```

Writing to keyed data queues

```
from("jms:queue:input")
.to("jt400://username:password@system/lib.lib/MSGINDQ.DTAQ?keyed=true");
```

Reading from keyed data queues

```
from("jt400://username:password@system/lib.lib/MSGOUTDQ.DTAQ?keyed=true&searchKey=MYKEY&searchType=GE")
```

```
.to("jms:queue:output");
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

LANGUAGE

Available as of Camel 2.5

The language component allows you to send Exchange to an endpoint which executes a script by any of the supported Languages in Camel.

By having a component to execute language scripts, it allows more dynamic routing capabilities. For example by using the Routing Slip or Dynamic Router EIPs you can send messages to language endpoints where the script is dynamic defined as well.

This component is provided out of the box in camel-core and hence no additional JARs is needed. You only have to include additional Camel components if the language of choice mandates it, such as using Groovy or JavaScript languages.

URI format

```
language://languageName[:script][?options]
```

And from Camel 2.11 onwards you can refer to an external resource for the script using same notation as supported by the other Languages in Camel

```
language://languageName:resource:scheme:location][?options]
```

URI Options

The component supports the following options.

Name	Default Value	Type	Description
languageName	null	String	The name of the Language to use, such as simple, groovy, javascript etc. This option is mandatory.
script	null	String	The script to execute.
transform	true	boolean	Whether or not the result of the script should be used as the new message body. By setting to false the script is executed but the result of the script is discarded.
contentcache	true	boolean	Camel 2.9: Whether to cache the script loaded from a resource. Note: from Camel 2.10.3 a cached script can be reloaded at runtime via JMX using the clearContentCache operation.
cachescript	false	boolean	Camel 2.13.2, 2.12.2/2.11.3: Whether to cache the compiled script. Turning this option on can gain performance as the script is only compiled/created once, and reuse when processing Camel messages. But this may cause side-effects with data left from previous evaluation spills into the next, and concurrency issues as well. If the script being evaluated is idempotent then this option can be turned on.
binary	false	boolean	Camel 2.14.1: Whether the script is binary content. This is intended to be used for loading resources using the Constant language, such as loading binary files.

Message Headers

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

Header	Description
CamelLanguageScript	The script to execute provided in the header. Takes precedence over script configured on the endpoint.

Examples

For example you can use the Simple language to Message Translator a message:

```
String script = URLEncoder.encode("Hello ${body}", "UTF-8");
from("direct:start").to("language:simple:" + script).to("mock:result");
```

In case you want to convert the message body type you can do this as well:

```
String script = URLEncoder.encode("${mandatoryBodyAs(String)}", "UTF-8");
from("direct:start").to("language:simple:" + script).to("mock:result");
```

You can also use the Groovy language, such as this example where the input message will be multiplied with 2:

```
String script = URLEncoder.encode("request.body * 2", "UTF-8");
from("direct:start").to("language:groovy:" + script).to("mock:result");
```

You can also provide the script as a header as shown below. Here we use XPath language to extract the text from the <foo> tag.

```
Object out = producer.requestBodyAndHeader("language>xpath", "<foo>Hello World</foo>", Exchange.LANGUAGE_SCRIPT, "/foo/text()");
assertEquals("Hello World", out);
```

Loading scripts from resources

Available as of Camel 2.9

You can specify a resource uri for a script to load in either the endpoint uri, or in the Exchange.LANGUAGE_SCRIPT header. The uri must start with one of the following schemes: file:, classpath:, or http:

For example to load a script from the classpath:

```
from("direct:start")
    // load the script from the classpath
    .to("language:simple:classpath:org/apache/camel/component/language/mysimplerest.txt")
    .to("mock:result");
```

By default the script is loaded once and cached. However you can disable the contentCache option and have the script loaded on each evaluation.

For example if the file myscript.txt is changed on disk, then the updated script is used:

```
from("direct:start")
    // the script will be loaded on each message, as we disabled cache
    .to("language:simple:file:target/script/myscript.txt?contentCache=false")
    .to("mock:result");
```

From Camel 2.11 onwards you can refer to the resource similar to the other Languages in Camel by prefixing with "resource:" as shown below:

```
from("direct:start")
    // load the script from the classpath
    .to("language:simple:resource:classpath:org/apache/camel/component/language/mysimplerest.txt")
    .to("mock:result");
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Languages
- Routing Slip
- Dynamic Router
- Script

LDAP COMPONENT

The **ldap** component allows you to perform searches in LDAP servers using filters as the message payload. This component uses standard JNDI (javax.naming package) to access the server.

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

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

URI format

```
ldap:ldapServerBean[?options]
```

The *ldapServerBean* portion of the URI refers to a DirContext bean in the registry. The LDAP component only supports producer endpoints, which means that an ldap URI cannot appear in the *from* at the start of a route.

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

Options

Name	Default Value	Description
base	ou=system	The base DN for searches
scope	subtree	Specifies how deeply to search the tree of entries, starting at the base DN. Value can be object, onelevel, or subtree.
pageSize	no paging used	Camel 2.6: When specified the ldap module uses paging to retrieve all results (most LDAP Servers throw an exception when trying to retrieve more than 1000 entries in one query). To be able to use this a LdapContext (subclass of DirContext) has to be passed in as ldapServerBean (otherwise an exception is thrown)
returnedAttributes	depends on LDAP Server	Camel 2.6: Comma-separated list of attributes that should be set in each entry of the result

(could be all or none)

Result

The result is returned in the Out body as a `ArrayList<javax.naming.directory.SearchResult>` object.

DirContext

The URL, `ldap://ldapserver`, references a Spring bean with the ID, `ldapserver`. The `ldapserver` bean may be defined as follows:

```
<bean id="ldapserver" class="javax.naming.directory.InitialDirContext" scope="prototype">
    <constructor-arg>
        <props>
            <prop key="java.naming.factory.initial">com.sun.jndi.ldap.LdapCtxFactory</prop>
            <prop key="java.naming.provider.url">ldap://localhost:10389</prop>
            <prop key="java.naming.security.authentication">none</prop>
        </props>
    </constructor-arg>
</bean>
```

The preceding example declares a regular Sun based LDAP `DirContext` that connects anonymously to a locally hosted LDAP server.

`DirContext` objects are **not** required to support concurrency by contract. It is therefore important that the directory context is declared with the setting, `scope="prototype"`, in the bean definition or that the context supports concurrency. In the Spring framework, prototype scoped objects are instantiated each time they are looked up.

Samples

Following on from the Spring configuration above, the code sample below sends an LDAP request to filter search a group for a member. The Common Name is then extracted from the response.

```
ProducerTemplate<Exchange> template = exchange
    .getContext().createProducerTemplate();

Collection<?> results = (Collection<?>) (template
    .sendBody(
        "ldap://ldapserver?base=ou=mygroup,ou=groups,ou=system",
        "(member=uid=huntc,ou=users,ou=system)");

if (results.size() > 0) {
    // Extract what we need from the device's profile

    Iterator<?> resultIter = results.iterator();
    SearchResult searchResult = (SearchResult) resultIter
        .next();
    Attributes attributes = searchResult
        .getAttributes();
    Attribute deviceCNAttr = attributes.get("cn");
    String deviceCN = (String) deviceCNAttr.get();

    ...
}
```

If no specific filter is required - for example, you just need to look up a single entry - specify a wildcard filter expression. For example, if the LDAP entry has a Common Name, use a filter expression like:

```
(cn=*)
```

Binding using credentials

A Camel end user donated this sample code he used to bind to the ldap server using credentials.

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
props.setProperty(Context.PROVIDER_URL, "ldap://localhost:389");
props.setProperty(Context.URL_PKG_PREFIXES, "com.sun.jndi.url");
props.setProperty(Context.REFERRAL, "ignore");
props.setProperty(Context.SECURITY_AUTHENTICATION, "simple");
props.setProperty(Context.SECURITY_PRINCIPAL, "cn=Manager");
props.setProperty(Context.SECURITY_CREDENTIALS, "secret");

SimpleRegistry reg = new SimpleRegistry();
reg.put("myldap", new InitialLdapContext(props, null));

CamelContext context = new DefaultCamelContext(reg);
context.addRoutes(
    new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:start").to("ldap:myldap?base=ou=test");
        }
    });
context.start();

ProducerTemplate template = context.createProducerTemplate();

Endpoint endpoint = context.getEndpoint("direct:start");
Exchange exchange = endpoint.createExchange();
```

```

exchange.getIn().setBody("(uid=test)");
Exchange out = template.send(endpoint, exchange);
Collection<SearchResult> data = out.getOut().getBody(Collection.class);
assert data != null;
assert !data.isEmpty();
System.out.println(out.getBody());
context.stop();

```

Configuring SSL

All required is to create a custom socket factory and reference it in the InitialDirContext bean - see below sample.

Listing 1. SSL Configuration

```

<xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
    http://camel.apache.org/schema/blueprint http://camel.apache.org/schema/blueprint/camel-blueprint.xsd">

    <sslContextParameters xmlns="http://camel.apache.org/schema/blueprint"
        id="sslContextParameters">
        <keyManagers
            keyPassword="{{keystore.pwd}}">
            <keyStore
                resource="{{keystore.url}}"
                password="{{keystore.pwd}}"/>
        </keyManagers>
    </sslContextParameters>

    <bean id="customSocketFactory" class="zotix.co.util.CustomSocketFactory">
        <argument ref="sslContextParameters" />
    </bean>
    <bean id="ldapserver" class="javax.naming.directory.InitialDirContext" scope="prototype">
        <argument>
            <props>
                <prop key="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory"/>
                <prop key="java.naming.provider.url" value="ldaps://lab.zotix.co:636"/>
                <prop key="java.naming.security.protocol" value="ssl"/>
                <prop key="java.naming.security.authentication" value="simple" />
                <prop key="java.naming.security.principal" value="cn=Manager,dc=example,dc=com"/>
                <prop key="java.naming.security.credentials" value="passw0rd"/>
                <prop key="java.naming.ldap.factory.socket"
                    value="zotix.co.util.CustomSocketFactory"/>
            </props>
        </argument>
    </bean>
</blueprint>

```

Listing 1. Custom Socket Factory

```

import org.apache.camel.util.jsse.SSLContextParameters;

import javax.net.SocketFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;
import java.io.IOException;
import java.net.InetAddress;
import java.net.Socket;
import java.security.KeyStore;

/** 
 * The CustomSocketFactory. Loads the KeyStore and creates an instance of SSLSocketFactory
 */
public class CustomSocketFactory extends SSLSocketFactory {

    private static SSLSocketFactory socketFactory;

    /**
     * Called by the getDefault() method.
     */
    public CustomSocketFactory() {
    }

    /**
     * Called by Blueprint DI to initialise an instance of SocketFactory
     * @param sslContextParameters
     */
    public CustomSocketFactory(SSLContextParameters sslContextParameters) {
        try {
            KeyStore keyStore = sslContextParameters.getKeyManagers().getKeyStore().createKeyStore();
            TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
            tmf.init(keyStore);
            SSLContext ctx = SSLContext.getInstance("TLS");
            ctx.init(null, tmf.getTrustManagers(), null);
            socketFactory = ctx.getSocketFactory();
        } catch (Exception ex) {
            ex.printStackTrace(System.err); /* handle exception */
        }
    }

    /**
     * Getter for the SocketFactory
     * @return
     */
    public static SocketFactory getDefault() {

```

```

    return new CustomSocketFactory();
}

@Override
public String[] getDefaultCipherSuites() {
    return socketFactory.getDefaultCipherSuites();
}

@Override
public String[] getSupportedCipherSuites() {
    return socketFactory.getSupportedCipherSuites();
}

@Override
public Socket createSocket(Socket socket, String string, int i, boolean bln) throws IOException {
    return socketFactory.createSocket(socket, string, i, bln);
}

@Override
public Socket createSocket(String string, int i) throws IOException {
    return socketFactory.createSocket(string, i);
}

@Override
public Socket createSocket(String string, int i, InetAddress ia, int i1) throws IOException {
    return socketFactory.createSocket(string, i, ia, i1);
}

@Override
public Socket createSocket(InetAddress ia, int i) throws IOException {
    return socketFactory.createSocket(ia, i);
}

@Override
public Socket createSocket(InetAddress ia, int i, InetAddress ia1, int i1) throws IOException {
    return socketFactory.createSocket(ia, i, ia1, i1);
}
}

```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

LOG COMPONENT

The `log:` component logs message exchanges to the underlying logging mechanism.

Camel uses `slf4j` which allows you to configure logging via, among others:

- Log4j
- Logback
- JDK Util Logging logging

URI format

```
log:loggingCategory[?options]
```

Where `LoggingCategory` is the name of the logging category to use. You can append query options to the URI in the following format, `?option=value&option=value&...`

Using Logger instance from the Registry

As of [Camel 2.12.4/2.13.1](#), if there's single instance of `org.slf4j.Logger` found in the Registry, the `loggingCategory` is no longer used to create logger instance. The registered instance is used instead. Also it is possible to reference particular Logger instance using `?logger=#myLogger` URI parameter. Eventually, if there's no registered and `URI` `logger` parameter, the logger instance is created using `loggingCategory`.

For example, a log endpoint typically specifies the logging level using the `level` option, as follows:

```
log:org.apache.camel.example?level=DEBUG
```

The default logger logs every exchange (*regular logging*). But Camel also ships with the `Throughput` logger, which is used whenever the `groupSize` option is specified.

Also a log in the DSL

There is also a log directly in the DSL, but it has a different purpose. Its meant for lightweight and human logs. See more details at [LogEIP](#).

Options

Option	Default	Type	Description
--------	---------	------	-------------

groupActiveOnly	true	boolean	If true, will hide stats when no new messages have been received for a time interval. If false, show stats regardless of message traffic.
groupDelay	0	Integer	Set the initial delay for stats (in millis).
groupInterval	null	Integer	If specified will group message stats by this time interval (in millis)
groupSize	null	Integer	An integer that specifies a group size for throughput logging.
level	INFO	String	Logging level to use. Possible values: ERROR, WARN, INFO, DEBUG, TRACE, OFF.
logLevel	LOG	String	Log level to use. Possible values: LOG, DEBUG, TRACE, INFO, WARN, ERROR, FATAL, OFF.
marker	null	String	Camel 2.9: An optional Marker name to use.

groupDelay and groupActiveOnly are only applicable when using groupInterval.

Formatting

The log formats the execution of exchanges to log lines.

By default, the log uses LogFormatter to format the log output, where LogFormatter has the following options:

Option	Default	Description
maxchars		Limits the number of characters logged per line. The default value, from Camel 2.9 is 10000.
multiline	false	If true, each piece of information is logged on a new line.
showAll	false	Quick option for turning all options on. If set to true, maxchars has to be manually set if to be used.
showBody		
showBodyType	true	Show the In body Java type.
showCaughtException	false	If the exchange has a caught exception, show the exception message (no stack trace). A stack trace is stored as a property on the exchange (using the key Exchange.EXCEPTION_CAUGHT) and for instance a docCatch can catch exceptions. See Try Catch Finally.
showException	false	If the exchange has an exception, show the exception message (no stack trace).
showExchangeId	false	Show the unique exchange ID.
showExchangePattern	true	Shows the Message Exchange Pattern (or MEP for short).
showFiles	false	Camel 2.9: Whether Camel should show file bodies or not, e.g., such as java.io.File.
showFuture	false	Whether Camel should show java.util.concurrent.Future bodies or not. If enabled Camel could potentially wait until the future task is done. Will not wait, by default.
showHeaders	false	Show the In message headers.
showOur	false	If the exchange has an Out message, show the our message.
showProperties	false	Show the properties.
showStackTrace	false	Show the stack trace, if an exchange has an exception. Only effective if one of showAll, showException or showCaughtException are enabled.
showStreams	false	Camel 2.8: Whether Camel should show stream bodies or not, e.g., such as java.io.InputStream. If you enable this, Camel will skip the stream body if it has already been read by this logger. To remedy this you will have to use Stream caching.
skipBodyLineSeparators	true	Camel 2.12.2: Whether to skip line separators when logging the message body. This will log the message body on a single line. Set to false to preserve any line separators present in the body, therefore logging the body as is.

Logging stream bodies

For older versions of Camel that do not support the showFiles or showStreams properties above, you can set the following property instead on the CamelContext to log both stream and file bodies:

```
camelContext.getProperties().put(Exchange.LOG_DEBUG_BODY_STREAMS, true);
```

Regular Logger Example

In the route below we log the incoming orders at DEBUG level before the order is processed:

```
from("activemq:orders")
.to("log:com.mycompany.order?level=DEBUG")
.to("bean:processOrder");
```

Or using Spring XML:

```
<route>
<from uri="activemq:orders"/>
<to uri="log:com.mycompany.order?level=DEBUG"/>
<to uri="bean:processOrder"/>
</route>
```

Regular Logger with Formatter

In the route below we log the incoming orders at INFO level before the order is processed.

```
from("activemq:orders")
.to("log:com.mycompany.order?showAll=true&multiline=true")
.to("bean:processOrder");
```

Throughput Logger With groupSize

In the route below we log the throughput of the incoming orders at DEBUG level grouped by 10 messages.

```
from("activemq:orders")
.to("log:com.mycompany.order?level=DEBUG&groupSize=10")
.to("bean:processOrder");
```

Throughput Logger With groupInterval

This route will result in message stats logged every 10s, with an initial 60s delay and stats should be displayed even if there isn't any message traffic.

```
from("activemq:orders")
    .to("log:com.mycompany.order?level=DEBUG&groupInterval=10000&groupDelay=60000&groupActiveOnly=false")
    .to("bean:processOrder");
```

The following will be logged:

```
"Received: 1000 new messages, with total 2000 so far. Last group took: 10000 millis which is: 100 messages per second. average: 100"
```

Full Customization of the Logged Output

Available as of Camel 2.11

With the options outlined in the #Formatting section, you can control much of the output of the logger. However, log lines will always follow this structure:

```
Exchange[Id:ID-machine-local-50656-1234567901234-1-2, ExchangePattern:InOut,
Properties:{CamelToEndpoint=log://org.apache.camel.component.log.TEST?showAll=true,
CamelCreatedTimestamp=Thu Mar 28 00:00:00 WET 2013},
Headers:{breadcrumbId=ID-machine-local-50656-1234567901234-1-1}, BodyType:String, Body>Hello World, Out: null]
```

This format is unsuitable in some cases, perhaps because you need to:

- Filter the headers and properties that are printed, to strike a balance between insight and verbosity.
- Adjust the log message to whatever you deem most readable.
- Tailor log messages for digestion by log mining systems, e.g. Splunk.
- Print specific body types differently.
- Etc.

Whenever you require absolute customization, you can create a class that implements the `ExchangeFormatter` interface. Within the `format(Exchange)` method you have access to the full Exchange, so you can select and extract the precise information you need, format it in a custom manner and return it. The return value will become the final log message.

You can have the Log component pick up your custom `ExchangeFormatter` in one of two ways:

Explicitly instantiating the LogComponent in your Registry

```
<bean name="log" class="org.apache.camel.component.log.LogComponent"
    <property name="exchangeFormatter" ref="myCustomFormatter"/>
</bean>
```

Convention Over Configuration

Simply by registering a bean with the name `logFormatter`; the Log Component is intelligent enough to pick it up automatically.

```
<bean name="logFormatter" class="com.xyz.MyCustomExchangeFormatter"/>
```

The `ExchangeFormatter` gets applied to **all Log endpoints within that Camel Context**. If you need a different `ExchangeFormatter` for each endpoint, just instantiate the `LogComponent` as many times as needed, and use the relevant bean name as the endpoint prefix.

From **Camel 2.11.2/2.12**: when using a custom log formatter, you can specify parameters in the log URI, which gets configured on the custom log formatter. Though when you do that you should define the `logFormatter` as prototype scoped so its not shared if you have different parameters.

Example:

```
<bean name="logFormatter" class="com.xyz.MyCustomExchangeFormatter" scope="prototype"/>
```

And then we can have Camel routes using the log URI with different options:

```
<to uri="log:foo?param1=foo&param2=100"/>
<!-- ... -->
<to uri="log:bar?param1=bar&param2=200"/>
```

Using Log Component in OSGi

Improvements from Camel 2.12.4/2.13.1

When using Log component inside OSGi (e.g., in Karaf), the underlying logging mechanisms are provided by PAX logging. It searches for a bundle which invokes `org.slf4j.LoggerFactory.getLogger()` method and associates the bundle with the logger instance. Without specifying custom `org.slf4j.Logger` instance, the logger created by Log component is associated with `camel-core` bundle.

In some scenarios it is required that the bundle associated with logger should be the bundle which contains route definition. To do this, either register a single instance of `org.slf4j.Logger` in the Registry or reference it using `logger URI` parameter.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Tracer
- How do I use log4j
- How do I use Java 1.4 logging
- LogEIP for using log directly in the DSL for human logs.

LUCENE (INDEXER AND SEARCH) COMPONENT

Available as of Camel 2.2

The **lucene** component is based on the Apache Lucene project. Apache Lucene is a powerful high-performance, full-featured text search engine library written entirely in Java. For more details about Lucene, please see the following links

- <http://lucene.apache.org/java/docs/>
- <http://lucene.apache.org/java/docs/features.html>

The lucene component in camel facilitates integration and utilization of Lucene endpoints in enterprise integration patterns and scenarios. The lucene component does the following

- builds a searchable index of documents when payloads are sent to the Lucene Endpoint
- facilitates performing of indexed searches in Camel

This component only supports producer endpoints.

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

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

URI format

```
lucene:searcherName:insert[?options]
lucene:searcherName:query[?options]
```

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

Insert Options

Name	Default Value	Description
analyzer	StandardAnalyzer	An Analyzer builds TokenStreams, which analyze text. It thus represents a policy for extracting index terms from text. The value for analyzer can be any class that extends the abstract class <code>org.apache.lucene.analysis.Analyzer</code> . Lucene also offers a rich set of analyzers out of the box.
indexdir	./indexdirectory	A file system directory in which index files are created upon analysis of the document by the specified analyzer
srcdir	null	An optional directory containing files to be used to be analyzed and added to the index at producer startup.

Query Options

Name	Default Value	Description
analyzer	StandardAnalyzer	An Analyzer builds TokenStreams, which analyze text. It thus represents a policy for extracting index terms from text. The value for analyzer can be any class that extends the abstract class <code>org.apache.lucene.analysis.Analyzer</code> . Lucene also offers a rich set of analyzers out of the box.
indexdir	./indexdirectory	A file system directory in which index files are created upon analysis of the document by the specified analyzer
maxhits	10	An integer value that limits the result set of the search operation.

Sending/Receiving Messages to/from the cache

Message Headers

Header	Description

QUERY The Lucene Query to performed on the index. The query may include wildcards and phrases.
RETURN_LUCENE_DOCS Camel 2.15: Set this header to true to include the actual Lucene documentation when returning hit information.

Lucene Producers

This component supports 2 producer endpoints.

- **insert** - The insert producer builds a searchable index by analyzing the body in incoming exchanges and associating it with a token ("content").
- **query** - The query producer performs searches on a pre-created index. The query uses the searchable index to perform score & relevance based searches. Queries are sent via the incoming exchange contains a header property name called 'QUERY'. The value of the header property 'QUERY' is a Lucene Query. For more details on how to create Lucene Queries check out http://lucene.apache.org/java/3_0_0/queryparsersyntax.html

Lucene Processor

There is a processor called LuceneQueryProcessor available to perform queries against lucene without the need to create a producer.

Lucene Usage Samples

Example 1: Creating a Lucene index

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start")
            .to("lucene:whiteSpaceQuotesIndex:insert?
                analyzer=#WhitespaceAnalyzer&indexDir=#whitespace&srcDir=#load_dir").
            to("mock:result");
    }
};
```

Example 2: Loading properties into the JNDI registry in the Camel Context

```
@Override
protected JndiRegistry createRegistry() throws Exception {
    JndiRegistry registry =
        new JndiRegistry(createJndiContext());
    registry.bind("whiteSpace", new File("./whitespaceIndexDir"));
    registry.bind("load_dir",
        new File("src/test/resources/sources"));
    registry.bind("WhitespaceAnalyzer",
        new WhitespaceAnalyzer());
    return registry;
}
...
CamelContext context = new DefaultCamelContext(createRegistry());
```

Example 2: Performing searches using a Query Producer

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start")
            .setHeader("QUERY", constant("Seinfeld"))
            .to("lucene:searchIndex:query?
                analyzer=#WhitespaceAnalyzer&indexDir=#whitespace&maxHits=20").
            to("direct:next");

        from("direct:next").process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                Hits hits = exchange.getIn().getBody(Hits.class);
                printResults(hits);
            }
        });
        private void printResults(Hits hits) {
            LOG.debug("Number of hits: " + hits.getNumberOfHits());
            for (int i = 0; i < hits.getNumberOfHits(); i++) {
                LOG.debug("Hit " + i + " Index Location: " + hits.getHit(i).getHitLocation());
                LOG.debug("Hit " + i + " Score: " + hits.getHit(i).getScore());
                LOG.debug("Hit " + i + " Data: " + hits.getHit(i).getData());
            }
        }
    }).to("mock:searchResult");
};
```

Example 3: Performing searches using a Query Processor

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

```

try {
    from("direct:start").
        setHeader("QUERY", constant("Rodney Dangerfield")),
        process(new LuceneQueryProcessor("target/stdindexDir", analyzer, null, 20)).
        to("direct:next");
} catch (Exception e) {
    e.printStackTrace();
}

from("direct:next").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        Hits hits = exchange.getIn().getBody(Hits.class);
        printResults(hits);
    }
}

private void printResults(Hits hits) {
    LOG.debug("Number of hits: " + hits.getNumberOfHits());
    for (int i = 0; i < hits.getNumberOfHits(); i++) {
        LOG.debug("Hit " + i + " Index Location:" + hits.getHit().get(i).getHitLocation());
        LOG.debug("Hit " + i + " Score:" + hits.getHit().get(i).getScore());
        LOG.debug("Hit " + i + " Data:" + hits.getHit().get(i).getData());
    }
}
}).to("mock:searchResult");
};

```

MAIL COMPONENT

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

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

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

We have discovered that the geronimo mail .jar (v1.6) has a bug when polling mails with attachments. It cannot correctly identify the Content-Type. So, if you attach a .jpeg file to a mail and you poll it, the Content-Type is resolved as text/plain and not as image/jpeg. For that reason, we have added an org.apache.camel.component.ContentResolver SPI interface which enables you to provide your own implementation and fix this bug by returning the correct Mime type based on the file name. So if the file name ends with jpeg/jpg, you can return image/jpeg.

You can set your custom resolver on the MailComponent instance or on the MailEndpoint instance.

POP3 or IMAP

POP3 has some limitations and end users are encouraged to use IMAP if possible.

Using mock-mail for testing

You can use a mock framework for unit testing, which allows you to test without the need for a real mail server. However you should remember to not include the mock-mail when you go into production or other environments where you need to send mails to a real mail server. Just the presence of the mock-javamail.jar on the classpath means that it will kick in and avoid sending the mails.

URI format

Mail endpoints can have one of the following URI formats (for the protocols, SMTP, POP3, or IMAP, respectively):

smtp://[username@[host[:port]][?options] pop3://[username@[host[:port]][?options] imap://[username@[host[:port]][?options]

The mail component also supports secure variants of these protocols (layered over SSL). You can enable the secure protocols by adding s to the scheme:

smtps://[username@[host[:port]][?options] pop3s://[username@[host[:port]][?options] imaps://[username@[host[:port]][?options]

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

Sample endpoints

Typically, you specify a URI with login credentials as follows (taking SMTP as an example):

smtp://[username@[host[:port]][?password=somepwd]

Alternatively, it is possible to specify both the user name and the password as query options:

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

For example:

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

DefaultPortsDefault ports

Default port numbers are supported. If the port number is omitted, Camel determines the port number to use based on the protocol.

confluenceTableSmall

Protocol	Default Port Number
SMTP	25
SMPTPS	465
POP3	110
POP3S	995

IMAP	143
IMAPS	993

Options

confluenceTableSmall

Property	Default	Description
host		The host name or IP address to connect to.
port	See #DefaultPorts	The TCP port number to connect on.
username		The user name on the email server.
password	null	The password on the email server.
ignoreURIScheme	false	If false, Camel uses the scheme to determine the transport (IMAP, SMTP etc.)
contentType	text/plain	The mail message content type. Use text/html for HTML messages.
folderName	INBOX	The folder to poll.
destination	username@host	@deprecated Use the to option instead. The TO recipient of the email.
to	username@host	The TO recipients (the receivers of the mail). Separate addresses with a comma. Email addresses containing such as "&" will need to be handled differently - see Handling special characters in email addresses .
replyTo	alias@host	As of Camel 2.8.4, 2.9.1+ , the Reply-To recipients (the response mail). Separate multiple email addresses with a comma.
cc	null	The CC recipients (the receivers of the mail). Separate addresses with a comma.
bcc	null	The BCC recipients (the receivers of the mail). Separate addresses with a comma.
from	camel@localhost	The FROM email address.
subject		As of Camel 2.3 , the Subject of the message being set in the header takes precedence over this option.
peek	true	Camel 2.11.3/2.12.2: Consumer only. Will mark the message as peeked before processing the mail message. This is only valid for IMAPMessage messages types only. By using peek the message will be marked as SEEN on the mail server, which allows reprocessing of the same message if there is an error processing in Camel.
delete	false	Deletes the messages after they have been processed setting the DELETED flag on the mail message. If false instead. As of Camel 2.10 you can override this configuration by setting a header with the key delete to determine if the message is deleted or not.
unseen	true	It is possible to configure a consumer endpoint so that unseen messages (that is, new messages) or all messages are consumed. Camel always skips deleted messages. The default option is to only unseen messages. POP3 does not support the unseen option is not supported in POP3; use IMAP instead. In fact, it is not in use if you also use searchTerm options. Instead, disable unseen when using searchTerm's then add searchTerm.unseen=false as a term.
copyTo	null	Camel 2.10: Consumer only. After processing a mail message, copy it to a mail folder with the given name. You can do this configuration value, with a header with the key copyTo to copy messages to folder names configured at runtime.
fetchsize	-1	Sets the maximum number of messages to consume at once. It can be used to avoid overloading a mail server, if a message contains a lot of messages. Default value of -1 means no fetch messages will be consumed. Setting the value to 0 is a special case, where Camel will not consume any messages at all.
alternativeBodyHeader	CamelMailAlternativeBody	Specifies the key to an IN message header that contains the alternative mail body. For example, if you send emails in text/html, you can provide an alternative mail body for non-HTML emails by using the alternative mail body with this key as a header.
debugMode	false	Enable debug mode on the underlying mail framework.

<code>connectionTimeout</code>	30000	framework logs the debug messages to <code>System.out</code> by default.
<code>consumer.initialDelay</code>	1000	The connection timeout in milliseconds. Default is 30 seconds.
<code>consumer.delay</code>	60000	Milliseconds before the polling starts.
<code>consumer.useFixedDelay</code>	false	Camel will poll the mailbox only once a minute by default, avoiding overloading the mail server.
<code>disconnect</code>	false	Set to true to use a fixed delay between polls, otherwise see <code>ScheduledExecutorService</code> in JDK for details.
<code>closeFolder</code>	true	Camel 2.8.3/2.9: Whether the consumer should disconnect after each poll. This forces Camel to connect on each poll.
<code>mail.xxx</code>	null	Camel 2.10.4: Whether the consumer should close the folder after each poll. Setting this option to false and having <code>disconnect=false</code> means the consumer keep the folder open between polls.
<code>mapMailMessage</code>	true	Set any additional Java Mail properties. For instance if you want to specify a special property when using POP3 you can now provide it in the URI such as: <code>mail.pop3.forgettopheaders=true</code> . You can also provide multiple such options, for example: <code>mail.pop3.forgettopheaders=true&mail.mime.encodeText=true</code> .
<code>maxMessagesPerPoll</code>	0	Camel 2.8: Specifies whether Camel should map the message to Camel body-Headers. If set to true, the body of the message is mapped to the body of the Camel IN message and headers are mapped to IN headers. If this option is set to false, the message contains a raw <code>javax.mail.Message</code> . You can then get the message body by calling <code>exchange.getIn().getBody(javax.mail.Message)</code> .
<code>javaMailSender</code>	null	Specifies the maximum number of messages to gather. If no maximum is set, no limit is set. Can be used to set a limit of e.g. 1 when downloading thousands of files when the server starts or negative to disable this option.
<code>ignoreUnsupportedCharset</code>	false	Specifies a pluggable <code>org.apache.camel.component.mail.MailBoxPostProcessor</code> instance in order to use a custom email implementation.
<code>searchTerm</code>	null	Option to let Camel ignore unsupported charset in the sending mails. If the charset is unsupported then <code>charsets</code> (which represents the unsupported charset) is removed from the message and it relies on the platform default instead.
<code>sslContextParameters</code>	null	Camel 2.10: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> instance. This reference overrides any configured <code>SSLContextParameters</code> component level. See Using the JSSE Configuration for more information.
<code>searchTerm.xxx</code>	null	Camel 2.11: Refers to a <code>javax.mail.search.SearchTerm</code> object. It filters mails based on search criteria such as subject, body, certain date etc. See further below for examples.
<code>sortTerm</code>	null	Camel 2.11: To configure search terms directly from the URI. It supports a limited number of terms defined by the <code>com.sun.mail.component.mail.SimpleSearch</code> class. See further below for examples.
<code>postProcessAction</code>	null	Camel 2.15: To configure the sort terms that IMAP supports. You may need to define an array of <code>com.sun.mail.imap.SortTerm</code> in the registry first and then use it in this URI option.
<code>skipFailedMessage</code>	false	Camel 2.16: You can also specify a comma separated list of sort terms on the URI that Camel will convert internally. For example, if you want to sort descending by date you would use <code>sortTerm=reversed</code> . So you can use any of the sort terms defined in <code>com.sun.mail.imap.SortTerm</code> .
<code>handleFailedMessage</code>	false	Camel 2.15.1: Refers to <code>org.apache.camel.component.mail.MailBoxPostProcessor</code> . It does post processing tasks on the mailbox once the processing is ended.
		Camel 2.15.1: If the mail consumer cannot retrieve a message, then this option allows to skip the message and move to the next mail message. The default behavior would be to throw an exception and no mails from the batch would be able to be processed by Camel.
		Camel 2.15.1: If the mail consumer cannot retrieve a message, then this option allows to handle the caused exception.

		error handler. By enable the bridge error handler on the Camel routing error handler can handle the exception behavior would be the consumer throws an exception the batch would be able to be routed by Camel.
dummyTrustManager	false	Camel 2.17: To use a dummy security setting for truststore. Should only be used for development mode, and not production.
idempotentRepository	null	Camel 2.17: A pluggable repository org.apache.camel.spi.IdempotentRepository which allows consuming from the same mailbox, and let the repository whether a mail message is valid for the consumer to process again.
idempotentRepositoryRemoveOnCommit	true	Camel 2.17: When using idempotent repository, then the message has been successfully processed and is consumed, the message id be removed from the idempotent repository in the repository. By default its assumed the message has no value to be kept in the repository, because the mail marked as seen/moved or deleted to prevent it from being processed again. And therefore having the message id stored in the repository has little value. However this option allows to keep the id, for whatever reason you may have.
mailUidGenerator		Camel 2.17: A pluggable MailUidGenerator that allows to generate UUID of the mail message.

SSL support

The underlying mail framework is responsible for providing SSL support. You may either configure SSL/TLS support by completely specifying the necessary Java Mail API configuration options, or you may provide a configured `SSLContextParameters` through the component or endpoint configuration.

Using the JSSE Configuration Utility

As of **Camel 2.10**, the mail component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the mail component.

Programmatic configuration of the endpoint

```
KeyStoreParameters ksp = new KeyStoreParameters(); ksp.setResource("/users/home/server/truststore.jks");
ksp.setPassword("keystorePassword"); TrustManagersParameters tmp = new TrustManagersParameters();
tmp.setKeyStore(ksp); SSLContextParameters scp = new SSLContextParameters(); scp.setTrustManagers(tmp); Registry
registry = ... registry.bind("sslContextParameters", scp); ... from(..) &nbsp; &nbsp; .to("smtps://smtp.google.com?
username=user@gmail.com&password=password&sslContextParameters=#sslContextParameters");
```

Spring DSL based configuration of endpoint

```
xm... <camel:sslContextParameters id="sslContextParameters"> <camel:trustManagers> <camel:keyStore
resource="/users/home/server/truststore.jks" password="keystorePassword"/> </camel:trustManagers>
</camel:sslContextParameters>... ...<to uri="smtps://smtp.google.com?
username=user@gmail.com&password=password&sslContextParameters=#sslContextParameters"/>...
```

Configuring JavaMail Directly

Camel uses SUN JavaMail, which only trusts certificates issued by well known Certificate Authorities (the default JVM trust configuration). If you issue your own certificates, you have to import the CA certificates into the JVM's Java trust/key store files, override the default JVM trust/key store files (see `SSLNOTES.txt` in JavaMail for details).

Mail Message Content

Camel uses the message exchange's IN body as the `MimeMessage` text content. The body is converted to `String.class`.

Camel copies all of the exchange's IN headers to the `MimeMessage` headers. You may wish to read How to avoid sending some or all message headers to prevent inadvertent data "leaks" from your application.

The subject of the `MimeMessage` can be configured using a header property on the IN message. The code below demonstrates this:
`{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mail/src/test/java/org/apache/camel/component/mail/MailSubjectTest.java}` The same applies for other `MimeMessage` headers such as recipients, so you can use a header property as `To:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mail/src/test/java/org/apache/camel/component/mail/MailUsingHeadersTest.java}` Since **Camel 2.11** When using the `MailProducer`

the send the mail to server, you should be able to get the message id of the MimeMessage with the key `CamelMailMessageId` from the Camel message header.

Headers take precedence over pre-configured recipients

The recipients specified in the message headers always take precedence over recipients pre-configured in the endpoint URI. The idea is that if you provide any recipients in the message headers, that is what you get. The recipients pre-configured in the endpoint URI are treated as a fallback.

In the sample code below, the email message is sent to `davsclaus@apache.org`, because it takes precedence over the pre-configured recipient, `info@mycompany.com`. Any cc and bcc settings in the endpoint URI are also ignored and those recipients will not receive any mail. The choice between headers and pre-configured settings is all or nothing: the mail component either takes the recipients exclusively from the headers or exclusively from the pre-configured settings. It is not possible to mix and match headers and pre-configured settings.

```
java Map<String, Object> headers = new HashMap<String, Object>(); headers.put("to", "davsclaus@apache.org");
template.sendBodyAndHeaders("smtp://admin@localhost?to=info@mycompany.com", "Hello World", headers);
```

Multiple recipients for easier configuration

It is possible to set multiple recipients using a comma-separated or a semicolon-separated list. This applies both to header settings and to settings in an endpoint URI. For example:

```
java Map<String, Object> headers = new HashMap<String, Object>(); headers.put("to", "davsclaus@apache.org";
jstrachan@apache.org ; ningjiang@apache.org");
```

The preceding example uses a semicolon, ;, as the separator character.

Setting sender name and email

You can specify recipients in the format, name <email>, to include both the name and the email address of the recipient.

For example, you define the following headers on the a Message:

```
Map headers = new HashMap(); map.put("To", "Claus Ibsen <davsclaus@apache.org>"); map.put("From", "James Strachan
<jstrachan@apache.org>"); map.put("Subject", "Camel is cool");
```

JavaMail API (ex SUN JavaMail)

JavaMail API is used under the hood for consuming and producing mails.

We encourage end-users to consult these references when using either POP3 or IMAP protocol. Note particularly that POP3 has a much more limited set of features than IMAP.

- JavaMail POP3 API
- JavaMail IMAP API
- And generally about the MAIL Flags

Samples

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

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

In the next sample, we poll a mailbox for new emails once every minute. Notice that we use the special consumer option for setting the poll interval, `consumer.delay`, as 60000 milliseconds = 60 seconds.

```
from("imap://admin@mymailserver.com password=secret&unseen=true&consumer.delay=60000") .to("seda://mails");
```

In this sample we want to send a mail to multiple recipients:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mail/src/test/java/org/apache/camel/component/mail/MailRecipientsTest.java}

Sending mail with attachment sample

Attachments are not support by all Camel components

The *Attachments API* is based on the Java Activation Framework and is generally only used by the Mail API. Since many of the other Camel components do not support attachments, the attachments could potentially be lost as they propagate along the route. The rule of thumb, therefore, is to add attachments just before sending a message to the mail endpoint.

The mail component supports attachments. In the sample below, we send a mail message containing a plain text message with a logo file attachment.{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mail/src/test/java/org/apache/camel/component/mail/MailAttachmentTest.java}

SSL sample

In this sample, we want to poll our Google mail inbox for mails. To download mail onto a local mail client, Google mail requires you to enable and configure SSL. This is done by logging into your Google mail account and changing your settings to allow IMAP access. Google have extensive documentation on how to do this.

```
from("imaps://imap.gmail.com?username=YOUR_USERNAME@gmail.com&password=YOUR_PASSWORD" +
"&delete=false&unseen=true&consumer.delay=60000").to("log:newmail");
The preceding route polls the Google mail inbox for new mails once every minute and logs the received messages to the newmail logger category.
Running the sample with DEBUG logging enabled, we can monitor the progress in the logs:
2008-05-08 06:32:09,640 DEBUG MailConsumer - Connecting to MailStore imaps://imap.gmail.com:993 (SSL enabled),
folder=INBOX 2008-05-08 06:32:11,203 DEBUG MailConsumer - Polling mailfolder: imaps://imap.gmail.com:993 (SSL enabled),
folder=INBOX 2008-05-08 06:32:11,640 DEBUG MailConsumer - Fetching 1 messages. Total 1 messages. 2008-05-08
06:32:12,171 DEBUG MailConsumer - Processing message: messageNumber=[332], from=[James Bond <007@mi5.co.uk>],
to=YOUR_USERNAME@gmail.com, subject=[... 2008-05-08 06:32:12,187 INFO newmail - Exchange[MailMessage:
messageNumber=[332], from=[James Bond <007@mi5.co.uk>], to=YOUR_USERNAME@gmail.com, subject=[...]
```

Consuming mails with attachment sample

In this sample we poll a mailbox and store all attachments from the mails as files. First, we define a route to poll the mailbox. As this sample is based on google mail, it uses the same route as shown in the SSL sample:

```
from("imaps://imap.gmail.com?username=YOUR_USERNAME@gmail.com&password=YOUR_PASSWORD" +
"&delete=false&unseen=true&consumer.delay=60000").process(new MyMailProcessor());
```

Instead of logging the mail we use a processor where we can process the mail from java code:

```
public void process(Exchange exchange) throws Exception { // the API is a bit clunky so we need to loop Map<String,
DataHandler> attachments = exchange.getIn().getAttachments(); if (attachments.size() > 0) { for (String name :
attachments.keySet()) { DataHandler dh = attachments.get(name); // get the file name String filename = dh.getName(); // get the
content and convert it to byte[] byte[] data = exchange.getContext().getTypeConverter().convertTo(byte[].class,
dh.getInputStream()); // write the data to a file FileOutputStream out = new FileOutputStream(filename); out.write(data);
out.flush(); out.close(); } }}
```

As you can see the API to handle attachments is a bit clunky but it's there so you can get the javax.activation.DataHandler so you can handle the attachments using standard API.

How to split a mail message with attachments

In this example we consume mail messages which may have a number of attachments. What we want to do is to use the Splitter EIP per individual attachment, to process the attachments separately. For example if the mail message has 5 attachments, we want the Splitter to process five messages, each having a single attachment. To do this we need to provide a custom Expression to the Splitter where we provide a List<Message> that contains the five messages with the single attachment.

The code is provided out of the box in Camel 2.10 onwards in the camel-mail component. The code is in the class: org.apache.camel.component.mail.SplitAttachmentsExpression, which you can find the source code here

In the Camel route you then need to use this Expression in the route as shown below:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mail/src/test/java/org/apache/camel/component/mail/MailSplitAttachmentsTest.java}If you use XML DSL then you need to declare a method call expression in the Splitter as shown below
```

```
xml<split> <method beanType="org.apache.camel.component.mail.SplitAttachmentsExpression"/> <to uri="mock:split"/> </split>
```

From Camel 2.16 onwards you can also split the attachments as byte[] to be stored as the message body. This is done by creating the expression with boolean true

```
SplitAttachmentsExpression split = SplitAttachmentsExpression(true);
```

And then use the expression with the splitter eip.

Using custom SearchTerm

Available as of Camel 2.11

You can configure a searchTerm on the MailEndpoint which allows you to filter out unwanted mails.

For example to filter mails to contain Camel in either Subject or Text you can do as follows:

```
xml<route> <from uri="imaps://mymailserver?username=foo&password=secret&searchTerm.subjectOrBody=Camel"/> <to
uri="bean:myBean"/> </route>
```

Notice we use the "searchTerm.subjectOrBody" as parameter key to indicate that we want to search on mail subject or body, to contain the word "Camel".

The class org.apache.camel.component.mail.SimpleSearchTerm has a number of options you can configure:

Or to get the new unseen emails going 24 hours back in time you can do. Notice the "now-24h" syntax. See the table below for more details.

```
xml<route> <from uri="imaps://mymailserver?username=foo&password=secret&searchTerm.fromSentDate=now-24h"/> <to
uri="bean:myBean"/> </route>
```

You can have multiple searchTerm in the endpoint uri configuration. They would then be combined together using AND operator, eg so both conditions must match. For example to get the last unseen emails going back 24 hours which has Camel in the mail subject you can do:

```
xml<route> <from uri="imaps://mymailserver?
username=foo&password=secret&searchTerm.subject=Camel&searchTerm.fromSentDate=now-24h"/> <to uri="bean:myBean"/>
</route> confluenceTableSmall
```

Option	Default	Description
unseen	true	Whether to limit by unseen mails only.
subjectOrBody	null	To limit by subject or body to contain the word.
subject	null	The subject must contain the word.
body	null	The body must contain the word.
from	null	The mail must be from a given email pattern.
to	null	The mail must be to a given email pattern.
fromSentDate	null	The mail must be sent after or equals (GE) a given date. The date pattern is yyyy-MM-dd HH:mm:ss, eg use "2012-01-01 00:00:00" to be from the year 2012 onwards. You can use "now" for current timestamp. The "now" syntax supports an optional offset, that can be specified as either + or - with a numeric value. For example for last 24 hours, you can use "now - 24h" or without spaces "now-24h". Notice that Camel supports shorthands for hours, minutes, and seconds.
toSentDate	null	The mail must be sent before or equals (BE) a given date. The date pattern is yyyy-MM-dd HH:mm:ss, eg use "2012-01-01 00:00:00" to be before the year 2012. You can use "now" for current timestamp. The "now" syntax supports an optional offset, that can be specified as either + or - with a numeric value. For example for last 24 hours, you can use "now - 24h" or without spaces "now-24h". Notice that Camel supports shorthands for hours, minutes, and seconds.

The `SimpleSearchTerm` is designed to be easily configurable from a POJO, so you can also configure it using a `<bean>` style in XML

```
<bean id="mySearchTerm" class="org.apache.camel.component.mail.SimpleSearchTerm"> <property name="subject" value="Order"/> <property name="to" value="acme-order@acme.com"/> <property name="fromSentDate" value="now"/> </bean>
```

You can then refer to this bean, using `#beanId` in your Camel route as shown:

```
xml<route> <from uri="imaps://mymailserver?username=foo&password=secret&searchTerm=#mySearchTerm"/> <to uri="bean:myBean"/> </route>
```

In Java there is a builder class to build compound `SearchTerm`s using the `{}{org.apache.camel.component.mail.SearchTermBuilder}` class.

This allows you to build complex terms such as:

```
// we just want the unseen mails which is not spam
SearchTermBuilder builder = new SearchTermBuilder();
builder.unseen().body(Op.not, "Spam").subject(Op.not, "Spam") // which was sent from either foo or bar
.from("foo@somewhere.com").from(Op.or, "bar@somewhere.com"); // .. and we could continue building the terms
term = builder.build();
```

Endpoint See Also

MINA COMPONENT

Deprecated

Deprecated

This component is deprecated as the Apache Mina 1.x project is EOL. Instead use MINA2 or Netty instead.

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

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

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

URI format

`mina:tcp://hostname[:port][?options]` `mina:udp://hostname[:port][?options]` `mina:vm://hostname[:port][?options]`

You can specify a codec in the Registry using the `codec` option. If you are using TCP and no codec is specified then the `textLine` flag is used to determine if text line based codec or object serialization should be used instead. By default the object serialization is used.

For UDP if no codec is specified the default uses a basic `bytebuffer` based codec.

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

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

In normal use, `camel-mina` only supports marshalling the body content—message headers and exchange properties are not sent.

However, the option, `transferExchange`, does allow you to transfer the exchange itself over the wire. See options below.

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

Options

confluenceTableSmall

Option	Default Value	Description
codec	null	You can refer to a named <code>ProtocolCodecFactory</code> instance in your Registry such as your Spring ApplicationContext, which is then used for the marshalling.
codec	null	You must use the # notation to look up your codec in the Registry. For example, use <code>#myCodec</code> to look up a bean with the id value, <code>myCodec</code> .
disconnect	false	Camel 2.3: Whether or not to disconnect(close) from Mina session right after use. Can be used for both consumer and producer.
textline	false	Only used for TCP. If no codec is specified, you can use this flag to indicate a text line based codec; if not specified or the value is false, then Object Serialization is assumed over TCP.
textlineDelimiter	DEFAULT	Only used for TCP and if <code>textline=true</code> . Sets the text line delimiter to use. Possible values are: DEFAULT, AUTO, WINDOWS, UNIX or MAC. If none provided, Camel will use DEFAULT. This delimiter is used to mark the end of text.
sync	true	Setting to set endpoint as one-way or request-response.
lazySessionCreation	true	Sessions can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started.
timeout	30000	You can configure the timeout that specifies how long to wait for a response from a remote server. The timeout unit is in milliseconds, so 60000 is 60 seconds. The timeout is only used for Mina producer.
encoding	JVM Default	You can configure the encoding (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. You can transfer the exchange over the wire instead of just the body. The following fields are transferred: In body, Out body, fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are <i>serializable</i> . Camel will exclude any non-serializable objects and log it at WARN level.
minaLogger	false	You can enable the Apache MINA logging filter. Apache MINA uses <code>slf4j</code> logging at INFO level to log all input and output.
filters	null	You can set a list of Mina IoFilters to register. The filters value must be one of the following: <ul style="list-style-type: none"> • Camel 2.2: comma-separated list of bean references (e.g. <code>#filterBean1,#filterBean2</code>) where each bean must be of type <code>org.apache.mina.common.IoFilter</code>. • before Camel 2.2: a reference to a bean of type <code>List<org.apache.mina.common.IoFilter></code>.
encoderMaxLineLength	-1	As of 2.1, you can set the textline protocol encoder max line length. By default the default value of Mina itself is used which are <code>Integer.MAX_VALUE</code> .
decoderMaxLineLength	-1	As of 2.1, you can set the textline protocol decoder max line length. By default the default value of Mina itself is used which are 1024.
producerPoolSize	16	The TCP producer is thread safe and supports concurrency much better. This option allows you to configure the number of threads in its thread pool for concurrent producers. Note: Camel has a pooled service which ensured it was already thread safe and supported concurrency already.
allowDefaultCodec	true	The mina component installs a default codec if both, codec is null and textline is false. Setting <code>allowDefaultCodec</code> to false prevents the mina component from installing a default codec as the first element in the filter chain. This is useful in scenarios where another filter must be the first in the filter chain, like the SSL filter.
disconnectOnNoReply	true	Camel 2.3: If sync is enabled then this option dictates MinaConsumer if it should disconnect where there is no reply to send back.
noReplyLogLevel	WARN	Camel 2.3: If sync is enabled this option dictates MinaConsumer which logging level to use when logging a there is no reply to send back. Values are: FATAL, ERROR, INFO, DEBUG, OFF.
clientMode	false	Camel 2.15: Consumer only. If the clientMode is true, mina consumer will connect the address as a TCP client.

Using a custom codec

See the Mina documentation how to write your own codec. To use your custom codec with camel-mina, you should register your codec in the Registry; for example, by creating a bean in the Spring XML file. Then use the codec option to specify the bean ID of your codec. See HL7 that has a custom codec.

Sample with sync=false

In this sample, Camel exposes a service that listens for TCP connections on port 6200. We use the `textline` codec. In our route, we create a Mina consumer endpoint that listens on port 6200:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mina/src/test/java/org/apache/camel/component/mina/MinaConsumerTest.java}
As the sample is part of a unit test, we test it by sending some data to it on port 6200.
{snippet:id=e2|lang=java|url=camel/trunk/components/camel-mina/src/test/java/org/apache/camel/component/mina/MinaConsumerTest.java}
```

Sample with sync=true

In the next sample, we have a more common use case where we expose a TCP service on port 6201 also use the `textline` codec. However, this time we want to return a response, so we set the `sync` option to true on the consumer.

```
{snippet:id=e3|lang=java|url=camel/trunk/components/camel-mina/src/test/java/org/apache/camel/component/mina/MinaConsumerTest.java}
```

Then we test the sample by sending some data and retrieving the response using the `template.requestBody()` method. As we know the response is a `String`, we cast it to `String` and can assert that the response is, in fact, something we have dynamically set in our processor code logic.

```
{snippet:id=e4|lang=java|url=camel/trunk/components/camel-mina/src/test/java/org/apache/camel/component/mina/MinaConsumerTest.java}
```

Sample with Spring DSL

Spring DSL can, of course, also be used for MINA. In the sample below we expose a TCP server on port 5555:

```
xml <route> <from uri="mina:tcp://localhost:5555?textline=true"/> <to uri="bean:myTCPOrderHandler"/> </route>
```

In the route above, we expose a TCP server on port 5555 using the `textline` codec. We let the Spring bean with ID, `myTCPOrderHandler`, handle the request and return a reply. For instance, the handler bean could be implemented as follows:

```
java public String handleOrder(String payload) { ... return "Order: OK" }
```

Configuring Mina endpoints using Spring bean style

Configuration of Mina endpoints is possible using regular Spring bean style configuration in the Spring DSL.

However, in the underlying Apache Mina toolkit, it is relatively difficult to set up the acceptor and the connector, because you can *not* use simple setters. To resolve this difficulty, we leverage the `MinaComponent` as a Spring factory bean to configure this for us. If you really need to configure this yourself, there are setters on the `MinaEndpoint` to set these when needed.

The sample below shows the factory approach:

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-mina/src/test/resources/org/apache/camel/component/mina/SpringMinaEndpointTest-context.xml}
```

And then we can refer to our endpoint directly in the route, as follows:

```
{snippet:id=e2|lang=xml|url=camel/trunk/components/camel-mina/src/test/resources/org/apache/camel/component/mina/SpringMinaEndpointTest-context.xml}
```

Closing Session When Complete

When acting as a server you sometimes want to close the session when, for example, a client conversion is finished. To instruct Camel to close the session, you should add a header with the key `CamelMinaCloseSessionWhenComplete` set to a boolean `true` value.

For instance, the example below will close the session after it has written the `bye` message back to the client:

```
java from("mina:tcp://localhost:8080?sync=true&textline=true").process(new Processor() { public void process(Exchange exchange) throws Exception { String body = exchange.getIn().getBody(String.class); exchange.getOut().setBody("Bye " + body); exchange.getOut().setHeader(MinaConstants.MINA_CLOSE_SESSION_WHEN_COMPLETE, true); } });
```

Get the IoSession for message

Available since Camel 2.1

You can get the `IoSession` from the message header with this key `MinaEndpoint.HEADER_MINA_IOSESSION`, and also get the local host address with the key `MinaEndpoint.HEADER_LOCAL_ADDRESS` and remote host address with the key `MinaEndpoint.HEADER_REMOTE_ADDRESS`.

Configuring Mina filters

Filters permit you to use some Mina Filters, such as `sslFilter`. You can also implement some customized filters. Please note that `codec` and `logger` are also implemented as Mina filters of type, `IoFilter`. Any filters you may define are appended to the end of the filter chain; that is, after `codec` and `logger`.

If using the `sslFilter` you need to add the `mina-filter-ssl` JAR to the classpath.

For instance, the example below will send a keep-alive message after 10 seconds of inactivity:

```

java
public class KeepAliveFilter extends IoFilterAdapter { @Override public void sessionCreated(NextFilter nextFilter, IoSession session) throws Exception { session.setIdleTime(IdleStatus.BOTH_IDLE, 10); nextFilter.sessionCreated(session); } @Override public void sessionIdle(NextFilter nextFilter, IoSession session, IdleStatus status) throws Exception { session.write("NOOP"); // NOOP is a FTP command for keep alive nextFilter.sessionIdle(session, status); } }

As Camel Mina may use a request-reply scheme, the endpoint as a client would like to drop some message, such as greeting when the connection is established. For example, when you connect to an FTP server, you will get a 220 message with a greeting (220 welcome to Pure-FTPd). If you don't drop the message, your request-reply scheme will be broken.

java
public class DropGreetingFilter extends IoFilterAdapter { @Override public void messageReceived(NextFilter nextFilter, IoSession session, Object message) throws Exception { if (message instanceof String) { String ftpMessage = (String) message; // "220" is given as greeting. "200 Zzz" is given as a response to "NOOP" (keep alive) if (ftpMessage.startsWith("220")) || or ftpMessage.startsWith("200 Zzz")) { // Dropping greeting return; } } nextFilter.messageReceived(session, message); } }

Then, you can configure your endpoint using Spring DSL:

xml
<bean id="myMinaFactory" class="org.apache.camel.component.mina.MinaComponent"> <constructor-arg index="0" ref="camelContext" /> </bean> <bean id="myMinaEndpoint" factory-bean="myMinaFactory" factory-method="createEndpoint"> <constructor-arg index="0" ref="myMinaConfig"/> </bean> <bean id="myMinaConfig" class="org.apache.camel.component.mina.MinaConfiguration"> <property name="protocol" value="tcp" /> <property name="host" value="localhost" /> <property name="port" value="2121" /> <property name="sync" value="true" /> <property name="minaLogger" value="true" /> <property name="filters" ref="listFilters"/> </bean> <bean id="listFilters" class="java.util.ArrayList" > <constructor-arg> <list value-type="org.apache.mina.common.IoFilter"> <bean class="com.example.KeepAliveFilter"/> <bean class="com.example.DropGreetingFilter"/> </list> </constructor-arg> </bean>

```

Endpoint See Also

- MINA2
- Netty

MOCK COMPONENT

Testing Summary Include

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 run, which typically fires messages to one or more endpoints, and finally the expectations can be asserted in a test case to ensure the system worked as expected.

This allows you to test various things like:

- The correct number of messages are received on each endpoint,
- The correct payloads are received, in the right order,
- Messages arrive on an endpoint in order, using some Expression to create an order testing function,
- Messages arrive match some kind of Predicate such as that specific headers have certain values, or that parts of the messages match some predicate, such as by evaluating an XPath or XQuery Expression.

Note that there is also the Test endpoint which is a Mock endpoint, but which uses a second endpoint to provide the list of expected message bodies and automatically sets up the Mock endpoint assertions. In other words, it's a Mock endpoint that automatically sets up its assertions from some sample messages in a File or database, for example.

Mock endpoints keep received Exchanges in memory indefinitely

Remember that Mock is designed for testing. When you add Mock endpoints to a route, each Exchange sent to the endpoint will be stored (to allow for later validation) in memory until explicitly reset or the JVM is restarted. If you are sending high volume and/or large messages, this may cause excessive memory use. If your goal is to test deployable routes inline, consider using NotifyBuilder or AdviceWith in your tests instead of adding Mock endpoints to routes directly.

From Camel 2.10 onwards there are two new options retainFirst, and retainLast that can be used to limit the number of messages the Mock endpoints keep in memory.

URI format

mock:someName[?options]

Where **someName** can be any string that uniquely identifies the endpoint.

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

Options

confluenceTableSmall

Option	Default	Description
reportGroup	null	A size to use a throughput logger for reporting
retainFirst		Camel 2.10: To only keep first X number of messages in memory.
retainLast		Camel 2.10: To only keep last X number of messages in memory.

Simple Example

Here's a simple example of Mock endpoint in use. First, the endpoint is resolved on the context. Then we set an expectation, and then, after the test has run, we assert that our expectations have been met.

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class);
resultEndpoint.expectedMessageCount(2); // send some messages ... // now lets assert that the mock:foo endpoint received 2
messages resultEndpoint.assertIsSatisfied();
```

You typically always call the `assertIsSatisfied()` method to test that the expectations were met after running a test.

Camel will by default wait 10 seconds when the `assertIsSatisfied()` is invoked. This can be configured by setting the `setResultWaitTime(millis)` method.

Using `assertPeriod`

Available as of Camel 2.7

When the assertion is satisfied then Camel will stop waiting and continue from the `assertIsSatisfied` method. That means if a new message arrives on the mock endpoint, just a bit later, that arrival will not affect the outcome of the assertion. Suppose you do want to test that no new messages arrives after a period thereafter, then you can do that by setting the `setAssertPeriod` method, for example:

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class); resultEndpoint.setAssertPeriod(5000);
resultEndpoint.expectedMessageCount(2); // send some messages ... // now lets assert that the mock:foo endpoint received 2
messages resultEndpoint.assertIsSatisfied();
```

Setting expectations

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

`confluenceTableSmall`

Method	Description
<code>expectedMessageCount(int)</code>	To define the expected message count on the endpoint.
<code>expectedMinimumMessageCount(int)</code>	To define the minimum number of expected messages on the endpoint.
<code>expectedBodiesReceived(...)</code>	To define the expected bodies that should be received (in order).
<code>expectedHeaderReceived(...)</code>	To define the expected header that should be received
<code>expectsAscending(Expression)</code>	To add an expectation that messages are received in order, using the given Expression to compare messages.
<code>expectsDescending(Expression)</code>	To add an expectation that messages are received in order, using the given Expression to compare messages.
<code>expectsNoDuplicates(Expression)</code>	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 <code>JMSMESSAGEID</code> 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 the following code:

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

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

Mocking existing endpoints

Available as of Camel 2.7

Camel now allows you to automatically mock existing endpoints in your Camel routes.

How it works

Important: The endpoints are still in action. What happens differently is that a Mock endpoint is injected and receives the message first and then delegates the message to the target endpoint. You can view this as a kind of intercept and delegate or endpoint listener.

Suppose you have the given route below:

```
{snippet:id=route1|title=Route|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/interceptor/AdviceWithMockEndpointsTest.java}
```

You can then use the `adviceWith` feature in Camel to mock all the endpoints in a given route from your unit test, as shown below:

```
{snippet:id=e1|title=adviceWith mocking all endpoints|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/interceptor/AdviceWithMockEndpointsTest.java}
```

Notice that the mock endpoints is given the uri `mock:<endpoint>`, for example `mock:direct:foo`. Camel logs at `INFO` level the endpoints being mocked:

INFO Advised endpoint [direct://foo] with mock endpoint [mock:direct:foo] Mocked endpoints are without parameters
 Endpoints which are mocked will have their parameters stripped off. For example the endpoint "log:foo?showAll=true" will be mocked to the following endpoint "mock:log:foo". Notice the parameters have been removed.
 Its also possible to only mock certain endpoints using a pattern. For example to mock all log endpoints you do as shown:
 {snippet:id=e2|lang=java|title=adviceWith mocking only log endpoints using a pattern|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/AdviceWithMockEndpointsTest.java}
 The pattern supported can be a wildcard or a regular expression. See more details about this at Intercept as its the same matching function used by Camel.
 Mind that mocking endpoints causes the messages to be copied when they arrive on the mock.
 That means Camel will use more memory. This may not be suitable when you send in a lot of messages.

Mocking existing endpoints using the camel-test component

Instead of using the advice with to instruct Camel to mock endpoints, you can easily enable this behavior when using the camel-test Test Kit.

The same route can be tested as follows. Notice that we return "*" from the isMockEndpoints method, which tells Camel to mock all endpoints.

If you only want to mock all log endpoints you can return "log*" instead.

```
{snippet:id=e1|lang=java|title=isMockEndpoints using camel-test kit|url=camel/trunk/components/camel-test/src/test/java/org/apache/camel/test/patterns/IsMockEndpointsJUnit4Test.java}
```

Mocking existing endpoints with XML DSL

If you do not use the camel-test component for unit testing (as shown above) you can use a different approach when using XML files for routes.

The solution is to create a new XML file used by the unit test and then include the intended XML file which has the route you want to test.

Suppose we have the route in the camel-route.xml file:

```
{snippet:id=e1|lang=xml|title=camel-route.xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/mock/camel-route.xml}
```

Then we create a new XML file as follows, where we include the camel-route.xml file and define a spring bean with the class org.apache.camel.impl.InterceptSendToMockEndpointStrategy which tells Camel to mock all endpoints:

```
{snippet:id=e1|lang=xml|title=test-camel-route.xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/mock/InterceptSendToMockEndpointStrategyTest.xml}
```

Then in your unit test you load the new XML file (test-camel-route.xml) instead of camel-route.xml.

To only mock all Log endpoints you can define the pattern in the constructor for the bean:

```
xml-bean id="mockAllEndpoints" class="org.apache.camel.impl.InterceptSendToMockEndpointStrategy"> <constructor-arg index="0" value="log"/> </bean>
```

Mocking endpoints and skip sending to original endpoint

Available as of Camel 2.10

Sometimes you want to easily mock and skip sending to a certain endpoints. So the message is detoured and send to the mock endpoint only. From Camel 2.10 onwards you can now use the mockEndpointsAndSkip method using AdviceWith or the Test Kit. The example below will skip sending to the two endpoints "direct:foo", and "direct:bar".

```
{snippet:id=e1|lang=java|title=adviceWith mock and skip sending to endpoints|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/AdviceWithMockMultipleEndpointsWithSkipTest.java}
```

The same example using the Test Kit

```
{snippet:id=e1|lang=java|title=isMockEndpointsAndSkip using camel-test kit|url=camel/trunk/components/camel-test/src/test/java/org/apache/camel/test/patterns/IsMockEndpointsAndSkipJUnit4Test.java}
```

Limiting the number of messages to keep

Available as of Camel 2.10

The Mock endpoints will by default keep a copy of every Exchange that it received. So if you test with a lot of messages, then it will consume memory.

From Camel 2.10 onwards we have introduced two options retainFirst and retainLast that can be used to specify to only keep N'th of the first and/or last Exchanges.

For example in the code below, we only want to retain a copy of the first 5 and last 5 Exchanges the mock receives.

```
MockEndpoint mock = getMockEndpoint("mock:data"); mock.setRetainFirst(5); mock.setRetainLast(5);  

mock.expectedMessageCount(2000); ... mock.assertIsSatisfied();
```

Using this has some limitations. The getExchanges() and getReceivedExchanges() methods on the MockEndpoint will return only the retained copies of the Exchanges. So in the example above, the list will contain 10 Exchanges; the first five, and the last five.

The retainFirst and retainLast options also have limitations on which expectation methods you can use. For example the expectedXXX methods that work on message bodies, headers, etc. will only operate on the retained messages. In the example above they can test only the expectations on the 10 retained messages.

Testing with arrival times

Available as of Camel 2.7

The Mock endpoint stores the arrival time of the message as a property on the Exchange.

```
Date time = exchange.getProperty(Exchange.RECEIVED_TIMESTAMP, Date.class);
```

You can use this information to know when the message arrived on the mock. But it also provides foundation to know the time interval between the previous and next message arrived on the mock. You can use this to set expectations using the `arrives` DSL on the Mock endpoint.

For example to say that the first message should arrive between 0-2 seconds before the next you can do:

```
mock.message(0).arrives().noLaterThan(2).seconds().beforeNext();
```

You can also define this as that 2nd message (0 index based) should arrive no later than 0-2 seconds after the previous:

```
mock.message(1).arrives().noLaterThan(2).seconds().afterPrevious();
```

You can also use `between` to set a lower bound. For example suppose that it should be between 1-4 seconds:

```
mock.message(1).arrives().between(1, 4).seconds().afterPrevious();
```

You can also set the expectation on all messages, for example to say that the gap between them should be at most 1 second:

```
mock.allMessages().arrives().noLaterThan(1).seconds().beforeNext();
```

time units

In the example above we use `seconds` as the time unit, but Camel offers `milliseconds`, and `minutes` as well.

Endpoint See Also

- Spring Testing
- Testing

MSV COMPONENT

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

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

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

Note that the Jing component also supports RelaxNG Compact Syntax

URI format

```
msv:someLocalOrRemoteResource[?options]
```

Where `someLocalOrRemoteResource` is some URL to a local resource on the classpath or a full URL to a remote resource or resource on the file system. For example

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

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

Options

Option	Default	Description
<code>useDOM</code>	<code>true</code>	Whether DOMSource/DOMResult or SaxSource/SaxResult should be used by the validator. Note: DOM must be used by the MSV component.

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://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <doTry>
            <to uri="msv:org/apache/camel/component/validator/msv/schema.rng"/>
            <to uri="mock:valid"/>

            <doCatch>
                <exception>org.apache.camel.ValidationException</exception>
                <to uri="mock:invalid"/>
            </doCatch>
            <doFinally>
                <to uri="mock:finally"/>
            </doFinally>
        </doTry>
    </route>
</camelContext>
```

```
</doTry>
</route>
</camelContext>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

MYBATIS

Available as of Camel 2.7

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

Maven users will need to add the following dependency to their pom.xml for this component:
`xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-mybatis</artifactId> <version>x.x.x</version> <!-- use the same version as your Camel core version --> </dependency>`

URI format

`mybatis:statementName[?options]`

Where **statementName** is the statement name in the MyBatis XML mapping file which maps to the query, insert, update or delete operation you wish to evaluate.

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

This component will by default load the MyBatis SqlMapConfig file from the root of the classpath with the expected name of `SqlMapConfig.xml`.

If the file is located in another location, you will need to configure the `configurationUri` option on the `MyBatisComponent` component.

Options

`confluenceTableSmall`

Option	Type	Default	Description
<code>consumer.onConsume</code>	String	null	Statements to run after consuming. Can be used, for example, to update rows after they have been consumed and processed in Camel. See sample later. Multiple statements can be separated with commas.
<code>consumer.useIterator</code>	boolean	true	If true each row returned when polling will be processed individually. If false the entire List of data is set as the IN body.
<code>consumer.routeEmptyResultSet</code>	boolean	false	Sets whether empty result sets should be routed.
<code>statementType</code>	StatementType	null	Mandatory to specify for the producer to control which kind of operation to invoke. The enum values are: <code>SelectOne</code> , <code>SelectList</code> , <code>Insert</code> , <code>InsertList</code> , <code>Update</code> , <code>UpdateList</code> , <code>Delete</code> , and <code>DeleteList</code> . Notice: <code>InsertList</code> is available as of Camel 2.10, and <code>updateList</code> , <code>deleteList</code> is available as of Camel 2.11.
<code>maxMessagesPerPoll</code>	int	0	This option is intended to split results returned by the database pool into the batches and deliver them in multiple exchanges. This integer defines the maximum messages to deliver in single exchange. By default, no maximum is set. Can be used to set a limit of e.g. 1000 to avoid when starting up the server that there are thousands of files. Set a value of 0 or negative to disable it.
<code>executorType</code>	String	null	Camel 2.11: The executor type to be used while executing statements. The supported values are: <code>simple</code> , <code>reuse</code> , <code>batch</code> . By default, the value is not specified and is equal to what MyBatis uses, i.e. <code>simple</code> . <code>simple</code> executor does nothing special. <code>reuse</code> executor reuses prepared statements. <code>batch</code> executor reuses statements and batches updates.
<code>outputHeader</code>	String	null	Camel 2.15: To store the result as a header instead of the message body. This allows to preserve the existing

			message body as-is.
inputHeader	String	null	Camel 2.15: "inputHeader" parameter to use a header value as input to the component instead of the body.
transacted	boolean	false	Camel 2.16.2: SQL consumer only: Enables or disables transaction. If enabled then if processing an exchange failed then the consumer break out processing any further exchanges to cause a rollback eager

Message Headers

Camel will populate the result message, either IN or OUT with a header with the statement used:
confluenceTableSmall

Header	Type	Description
CamelMyBatisStatementName	String	The statementName used (for example: insertAccount).
CamelMyBatisResult	object	The response returned from MtBatis in any of the operations. For instance an INSERT could return the auto-generated key, or number of rows etc.

Message Body

The response from MyBatis will only be set as the body if it's a SELECT statement. That means, for example, for INSERT statements Camel will not replace the body. This allows you to continue routing and keep the original body. The response from MyBatis is always stored in the header with the key CamelMyBatisResult.

Samples

For example if you wish to consume beans from a JMS queue and insert them into a database you could do the following:
from("activemq:queue:newAccount").to("mybatis:insertAccount?statementType=Insert");

Notice we have to specify the **statementType**, as we need to instruct Camel which kind of operation to invoke.

Where **insertAccount** is the MyBatis ID in the SQL mapping file:

```
xml <!-- Insert example, using the Account parameter class --> <insert id="insertAccount" parameterType="Account"> insert into ACCOUNT ( ACC_ID, ACC_FIRST_NAME, ACC_LAST_NAME, ACC_EMAIL ) values ( #{id}, #{firstName}, #{lastName}, # {emailAddress} ) </insert>
```

Using StatementType for better control of MyBatis

When routing to an MyBatis endpoint you will want more fine grained control so you can control whether the SQL statement to be executed is a SELECT, UPDATE, DELETE or INSERT etc. So for instance if we want to route to an MyBatis endpoint in which the IN body contains parameters to a SELECT statement we can do:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mybatis/src/test/java/org/apache/camel/component/mybatis/MyBatisSelectOneTest.java}In the code above we can invoke the MyBatis statement selectAccountById and the IN body should contain the account id we want to retrieve, such as an Integer type.

We can do the same for some of the other operations, such as **selectList**:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mybatis/src/test/java/org/apache/camel/component/mybatis/MyBatisSelectListTest.java}And the same for UPDATE, where we can send an Account object as the IN body to MyBatis:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mybatis/src/test/java/org/apache/camel/component/mybatis/MyBatisUpdateTest.java}
```

Using InsertList StatementType

Available as of Camel 2.10

MyBatis allows you to insert multiple rows using its for-each batch driver. To use this, you need to use the <foreach> in the mapper XML file. For example as shown below:{snippet:id=insertList|lang=xml|url=camel/trunk/components/camel-mybatis/src/test/resources/org/apache/camel/component/mybatis/Account.xml}Then you can insert multiple rows, by sending a Camel message to the mybatis endpoint which uses the **InsertList** statement type, as shown below:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mybatis/src/test/java/org/apache/camel/component/mybatis/MyBatisInsertListTest.java}
```

Using UpdateList StatementType

Available as of Camel 2.11

MyBatis allows you to update multiple rows using its for-each batch driver. To use this, you need to use the <foreach> in the mapper XML file. For example as shown below:

```
xml<update id="batchUpdateAccount" parameterType="java.util.Map"> update ACCOUNT set ACC_EMAIL = #{emailAddress} where ACC_ID in <foreach item="Account" collection="list" open="(" close=")" separator=" "> #{Account.id} </foreach> </update>
```

Then you can update multiple rows, by sending a Camel message to the mybatis endpoint which uses the `UpdateList` statement type, as shown below:

```
from("direct:start") .to("mybatis:batchUpdateAccount?statementType=UpdateList") .to("mock:result");
```

Using DeleteList StatementType

Available as of Camel 2.11

MyBatis allows you to delete multiple rows using its for-each batch driver. To use this, you need to use the `<foreach>` in the mapper XML file. For example as shown below:

```
xml-delete id="batchDeleteAccountById" parameterType="java.util.List"> delete from ACCOUNT where ACC_ID in <foreach item="AccountID" collection="list" open="(" close=")" separator=","> #{AccountID} </foreach> </delete>
Then you can delete multiple rows, by sending a Camel message to the mybatis endpoint which uses the DeleteList statement type, as shown below:
```

```
from("direct:start") .to("mybatis:batchDeleteAccount?statementType=DeleteList") .to("mock:result");
```

Notice on InsertList, UpdateList and DeleteList StatementTypes

Parameter of any type (List, Map, etc.) can be passed to mybatis and an end user is responsible for handling it as required with the help of mybatis dynamic queries capabilities.

Scheduled polling example

This component supports scheduled polling and can therefore be used as a Polling Consumer. For example to poll the database every minute:

```
from("mybatis:selectAllAccounts?delay=60000").to("activemq:queue:allAccounts");
```

See "ScheduledPollConsumer Options" on Polling Consumer for more options.

Alternatively you can use another mechanism for triggering the scheduled polls, such as the Timer or Quartz components. In the sample below we poll the database, every 30 seconds using the Timer component and send the data to the JMS queue:

```
javafrom("timer://pollTheDatabase?delay=30000").to("mybatis:selectAllAccounts").to("activemq:queue:allAccounts");
```

And the MyBatis SQL mapping file used:

```
xml <!-- Select with no parameters using the result map for Account class. --> <select id="selectAllAccounts" resultMap="AccountResult"> select * from ACCOUNT </select>
```

Using onConsume

This component supports executing statements **after** data have been consumed and processed by Camel. This allows you to do post updates in the database. Notice all statements must be UPDATE statements. Camel supports executing multiple statements whose names should be separated by commas.

The route below illustrates we execute the `consumeAccount` statement data is processed. This allows us to change the status of the row in the database to processed, so we avoid consuming it twice or more.

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-mybatis/src/test/java/org/apache/camel/component/mybatis/MyBatisQueueTest.java}And the statements in the sqlmap file:  
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-mybatis/src/test/resources/org/apache/camel/component/mybatis/Account.xml}  
{snippet:id=e2|lang=xml|url=camel/trunk/components/camel-mybatis/src/test/resources/org/apache/camel/component/mybatis/Account.xml}
```

Participating in transactions

Setting up a transaction manager under camel-mybatis can be a little bit fiddly, as it involves externalising the database configuration outside the standard MyBatis `SqlMapConfig.xml` file.

The first part requires the setup of a `DataSource`. This is typically a pool (either DBCP, or c3p0), which needs to be wrapped in a Spring proxy. This proxy enables non-Spring use of the `DataSource` to participate in Spring transactions (the MyBatis `SqlSessionFactory` does just this).

```
xml <bean id="dataSource" class="org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy"> <constructor-arg> <bean class="com.mchange.v2.c3p0.ComboPooledDataSource"> <property name="driverClass" value="org.postgresql.Driver"/> <property name="jdbcUrl" value="jdbc:postgresql://localhost:5432/myDatabase"/> <property name="user" value="myUser"/> <property name="password" value="myPassword"/> </bean> </constructor-arg> </bean>
```

This has the additional benefit of enabling the database configuration to be externalised using property placeholders.

A transaction manager is then configured to manage the outermost `DataSource`:

```
xml <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"> <property name="dataSource" ref="dataSource"/> </bean>
```

A mybatis-spring `SqlSessionFactoryBean` then wraps that same `DataSource`:

```
xml <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"> <property name="dataSource" ref="dataSource"/> <!-- standard mybatis config file --> <property name="configLocation" value="/META-INF/SqlMapConfig.xml"/> <!-- externalised mappers --> <property name="mapperLocations" value="classpath*:META-INF/mappers/**/*.*xml"/> </bean>
```

The camel-mybatis component is then configured with that factory:

```
xml <bean id="mybatis" class="org.apache.camel.component.mybatis.MyBatisComponent"> <property name="sqlSessionFactory" ref="sqlSessionFactory"/> </bean>
Finally, a transaction policy is defined over the top of the transaction manager, which can then be used as usual:
xml <bean id="PROPAGATION_REQUIRED" class="org.apache.camel.spi.SpringTransactionPolicy"> <property name="transactionManager" ref="txManager"/> <property name="propagationBehaviorName" value="PROPAGATION_REQUIRED"/> </bean> <camelContext id="my-model-context" xmlns="http://camel.apache.org/schema/spring"> <route id="insertModel"> <from uri="direct:insert"/> <transacted ref="PROPAGATION_REQUIRED"/> <to uri="mybatis:myModel.insert?statementType=Insert"/> </route> </camelContext>
Endpoint See Also
```

NAGIOS

Available as of Camel 2.3

The Nagios component allows you to send passive checks to Nagios.

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

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

URI format

```
nagios://host[:port][?Options]
```

Camel provides two abilities with the Nagios component. You can send passive check messages by sending a message to its endpoint.

Camel also provides a EventNotifier which allows you to send notifications to Nagios.

Options

Name	Default Value	Description
host	none	This is the address of the Nagios host where checks should be send.
port	none	The port number of the host.
password	none	Password to be authenticated when sending checks to Nagios.
connectionTimeout	5000	Connection timeout in millis.
timeout	5000	Sending timeout in millis.
nagiosSettings	none	This is the already configured camel-nagios (endpoint) core NagiosSettings object. Then any of the other options are not in use, if using this.
sendsync	true	Use syncronous mode when sending a passive check. Setting it to false will allow Camel to continue routing the message and the passive check message will be send asynchronously.
encryptionMethod	no	Camel 2.9 To specify an encryption method. Possible values: no, xor, or tripledes.

Headers

Name	Description
CamelNagiosHostName	This is the address of the Nagios host where checks should be send. This header will override any existing hostname configured on the endpoint.
CamelNagiosServiceName	The service name. Will default to the CamelContext name. Camel will use this header to identify the service being checked.
CamelNagiosServiceName	The service name. Will default to the CamelContext name.

Sending message examples

You can send a message to Nagios where the message payload contains the message. By default it will be ok level and use the CamelContext name as the service name. You can overrule these values using headers as shown above.

For example we send the Hello Nagios message to Nagios as follows:

```
template.sendBody("direct:start", "Hello Nagios");
from("direct:start").to("nagios:127.0.0.1:5667?password=secret").to("mock:result");
```

To send a CRITICAL message you can send the headers such as:

```
Map headers = new HashMap();
headers.put(NagiosConstants.LEVEL, "CRITICAL");
headers.put(NagiosConstants.HOST_NAME, "myHost");
headers.put(NagiosConstants.SERVICE_NAME, "myService");
template.sendBodyAndHeaders("direct:start", "Hello Nagios", headers);
```

Using NagiosEventNotifier

The Nagios component also provides an EventNotifier which you can use to send events to Nagios. For example we can enable this from Java as follows:

```
NagiosEventNotifier notifier = new NagiosEventNotifier();
notifier.getConfiguration().setHost("localhost");
notifier.getConfiguration().setPort(5667);
notifier.getConfiguration().setPassword("password");

CamelContext context = ...
context.getManagementStrategy().addEventNotifier(notifier);
return context;
```

In Spring XML its just a matter of defining a Spring bean with the type EventNotifier and Camel will pick it up as documented here: Advanced configuration of CamelContext using Spring.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

NETTY COMPONENT

Available as of Camel 2.3

This component is deprecated. You should use Netty4.

The **netty** component in Camel is a socket communication component, based on the Netty project.

Netty is a NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients.

Netty greatly simplifies and streamlines network programming such as TCP and UDP socket server.

This camel component supports both producer and consumer endpoints.

The Netty component has several options and allows fine-grained control of a number of TCP/UDP communication parameters (buffer sizes, keepAlives, tcpNoDelay etc) and facilitates both In-Only and In-Out communication on a Camel route.

Maven users will need to add the following dependency to their pom.xml for this component:
`xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-netty</artifactId> <version>x.x.x</version> <!-- use the same version as your Camel core version --> </dependency>`

URI format

The URI scheme for a netty component is as follows

`netty:tcp://localhost:99999[?options] netty:udp://remotehost:99999[?options]`

This component supports producer and consumer endpoints for both TCP and UDP.

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

Options

confluenceTableSmall

Name	Default Value	Description
keepAlive	true	Setting to ensure socket is not closed due to inactivity
tcpNoDelay	true	Setting to improve TCP protocol performance
backlog		Camel 2.9.6/2.10.4/2.11: Allows to configure a backlog for netty Note the backlog is just a best effort depending on the OS. Setting value such as 200, 500 or 1000, tells the TCP stack how long the be. If this option is not configured, then the backlog depends on
broadcast	false	Setting to choose Multicast over UDP
connectTimeout	10000	Time to wait for a socket connection to be available. Value is in milliseconds
reuseAddress	true	Setting to facilitate socket multiplexing
sync	true	Setting to set endpoint as one-way or request-response
synchronous	false	Camel 2.10: Whether Asynchronous Routing Engine is not in use. If the Asynchronous Routing Engine is used, true to force processing
ssl	false	Setting to specify whether SSL encryption is applied to this endpoint
sslClientCertHeaders	false	Camel 2.12: When enabled and in SSL mode, then the Netty component will add Camel Message with headers having information about the client subject name, issuer name, serial number, and the valid date range

sendBufferSize	65536 bytes	The TCP/UDP buffer sizes to be used during outbound communication.
receiveBufferSize	65536 bytes	The TCP/UDP buffer sizes to be used during inbound communication.
option.xxx	null	Camel 2.11/2.10.4: Allows to configure additional netty options under the option prefix. For example "option.child.keepAlive=false" to set the netty child.keepAlive=false". See the Netty documentation for possible values.
corePoolSize	10	The number of allocated threads at component startup. Defaults to 10. This option is removed from Camel 2.9.2 onwards. As we rely on Netty's thread pool, this is no longer needed.
maxPoolsize	100	The maximum number of threads that may be allocated to this exchange. Defaults to 100. Note: This option is removed from Camel 2.9.2 onwards. A default setting of 100 is used.
disconnect	false	Whether or not to disconnect(close) from Netty Channel right after both consumer and producer.
lazyChannelCreation	true	Channels can be lazily created to avoid exceptions, if the remote host is running when the Camel producer is started.
transferExchange	false	Only used for TCP. You can transfer the exchange over the wire body. The following fields are transferred: In body, Out body, fault body, Out headers, fault headers, exchange properties, exchange exception. Note that the objects are serializable. Camel will exclude any non-serializable objects at WARN level.
disconnectOnNoReply	true	If sync is enabled then this option dictates NettyConsumer if it should disconnect where there is no reply to send back.
noReplyLogLevel	WARN	If sync is enabled this option dictates NettyConsumer which logging level to use when there is no reply to send back. Values are: FATAL, ERROR, WARN, INFO, DEBUG, TRACE, OFF.
serverExceptionCaughtLogLevel	WARN	Camel 2.11.1: If the server (NettyConsumer) catches an exception, then this option dictates the logging level.
serverClosedChannelExceptionCaughtLogLevel	DEBUG	Camel 2.11.1: If the server (NettyConsumer) catches an java.nio.channels.ClosedChannelException then its logged at this level. This is used to avoid logging the closed channel exception when disconnecting abruptly and then cause a flood of closed exceptions in the logs.
allowDefaultCodec	true	Camel 2.4: The netty component installs a default codec if both, null and textline is false. Setting allowDefaultCodec to false prevents the component from installing a default codec as the first element in the pipeline.
textline	false	Camel 2.4: Only used for TCP. If no codec is specified, you can indicate a text line based codec; if not specified or the value is false, then the default codec is used. If true, then the textline codec is used. Serialization is assumed over TCP.
delimiter	LINE	Camel 2.4: The delimiter to use for the textline codec. Possible values are LINE, CRLF, CR, LF, and NULL.
decoderMaxLineLength	1024	Camel 2.4: The max line length to use for the textline codec.
autoAppendDelimiter	true	Camel 2.4: Whether or not to auto append missing end delimiter for the textline codec.
encoding	null	Camel 2.4: The encoding (a charset name) to use for the textline codec. If not specified, Camel will use the JVM default Charset.
workerCount	null	Camel 2.9: When netty works on nio mode, it uses default workerCount from Netty, which is <code>cpu_core_threads*2</code> . User can use this option to override the default workerCount from Netty.
sslContextParameters	null	Camel 2.9: SSL configuration using an org.apache.camel.util.jsse.SSLContextParameters instance. See the JSSE Configuration Utility.
receiveBufferSizePredictor	null	Camel 2.9: Configures the buffer size predictor. See details at JavaDoc and this mail thread.
requestTimeout	0	Camel 2.11.1: Allows to use a timeout for the Netty producer when sending to the server. By default no timeout is in use. The value is in milliseconds. The requestTimeout is using Netty's ReadTimeoutHandler. Camel 2.16, 2.15.3 you can also override this setting by setting the header <code>CamelNettyRequestTimeout</code> .
needClientAuth	false	Camel 2.11: Configures whether the server needs client authentication for SSL.
orderedThreadPoolExecutor	true	Camel 2.10.2: Whether to use ordered thread pool, to ensure exchanges are processed orderly on the same channel. See details at the netty javadoc of <code>OrderedThreadPoolExecutor</code> .

		org.jboss.netty.handler.execution.OrderedMemoryAware for more details.
maximumPoolSize	16	Camel 2.10.2: The core pool size for the ordered thread pool, if Since Camel 2.14.1: This option is move the NettyComponen
producerPoolEnabled	true	Camel 2.10.4/Camel 2.11: Producer only. Whether producer po Important: Do not turn this off, as the pooling is needed for han
producerPoolMaxActive	-1	Camel 2.10.3: Producer only. Sets the cap on the number of obj allocated by the pool (checked out to clients, or idle awaiting che time. Use a negative value for no limit.
producerPoolMinIdle	0	Camel 2.10.3: Producer only. Sets the minimum number of insta producer pool before the evictor thread (if active) spawns new o
producerPoolMaxIdle	100	Camel 2.10.3: Producer only. Sets the cap on the number of "idle pool.
producerPoolMinEvictableIdle	300000	Camel 2.10.3: Producer only. Sets the minimum amount of time object may sit idle in the pool before it is eligible for eviction by t
bootstrapConfiguration	null	Camel 2.12: Consumer only. Allows to configure the Netty Server using a org.apache.camel.component.netty.NettyServerBootstrap instance. This can be used to reuse the same configuration for r align their configuration more easily.
bossPoll	null	Camel 2.12: To use a explicit org.jboss.netty.channel.sock the boss thread pool. For example to share a thread pool with m default each consumer has their own boss pool with 1 core thre
workerPool	null	Camel 2.12: To use a explicit org.jboss.netty.channel.sock as the worker thread pool. For example to share a thread pool w consumers. By default each consumer has their own worker poc core threads.
channelGroup	null	Camel 2.17 To use a explicit io.netty.channel.group.ChannelGr to broadcast a message to multiple channels.
networkInterface	null	Camel 2.12: Consumer only. When using UDP then this option c a network interface by its name, such as eth0 to join a multicast
udpConnectionlessSending	false	Camel 2.15: Producer only. This option supports connection less is a real fire and forget. A connected udp send receive the PortL if no one is listen on the receiving port.
clientMode	false	Camel 2.15: Consumer only. If the clientMode is true, netty cor the address as a TCP client.
useChannelBuffer	false	Camel 2.16: Producer only. If the usechannelBuffer is true, ne the message body into channelBuffer before sending it out.

Registry based Options

Codec Handlers and SSL Keystores can be enlisted in the Registry, such as in the Spring XML file.

The values that could be passed in, are the following:
confluenceTableSmall

Name	Description
passphrase	password setting to use in order to encrypt/decrypt payloads sent using SSH
keyStoreFormat	keystore format to be used for payload encryption. Defaults to "JKS" if not set
securityProvider	Security provider to be used for payload encryption. Defaults to "SunX509" if not set.
keyStoreFile	deprecated: Client side certificate keystore to be used for encryption
truststoreFile	deprecated: Server side certificate keystore to be used for encryption
keyStoreResource	Camel 2.11.1: Client side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with "classpath:", "file:", or "http:" to load the resource from different systems.
trustStoreResource	Camel 2.11.1: Server side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with "classpath:", "file:", or "http:" to load the resource from different systems.
sslHandler	Reference to a class that could be used to return an SSL Handler
encoder	A custom channelHandler class that can be used to perform special marshalling of outbound payloads. Must override org.jboss.netty.channel.ChannelDownStreamHandler.
encoders	A list of encoders to be used. You can use a String which have values separated by comma, and

have the values be looked up in the Registry. Just remember to prefix the value with # so Camel knows it should lookup.

decoder	A custom <code>ChannelHandler</code> class that can be used to perform special marshalling of inbound payloads. Must override <code>org.jboss.netty.channel.ChannelUpstreamHandler</code> .
decoders	A list of decoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with # so Camel knows it should lookup.

Important: Read below about using non shareable encoders/decoders.

Using non shareable encoders or decoders

If your encoders or decoders is not shareable (eg they have the `@Shareable` class annotation), then your encoder/decoder must implement the `org.apache.camel.component.netty.ChannelHandlerFactory` interface, and return a new instance in the `newChannelHandler` method. This is to ensure the encoder/decoder can safely be used. If this is not the case, then the Netty component will log a WARN when an endpoint is created.

The Netty component offers a `org.apache.camel.component.netty.ChannelHandlerFactories` factory class, that has a number of commonly used methods.

Sending Messages to/from a Netty endpoint

Netty Producer

In Producer mode, the component provides the ability to send payloads to a socket endpoint using either TCP or UDP protocols (with optional SSL support).

The producer mode supports both one-way and request-response based operations.

Netty Consumer

In Consumer mode, the component provides the ability to:

- listen on a specified socket using either TCP or UDP protocols (with optional SSL support),
- receive requests on the socket using text/xml, binary and serialized object based payloads and
- send them along on a route as message exchanges.

The consumer mode supports both one-way and request-response based operations.

Headers

The following headers are filled for the exchanges created by the Netty consumer:
confluenceTableSmall

Header key	Class	Description
<code>NettyConstants.NETTY_CHANNEL_HANDLER_CONTEXT</code> / <code>CamelNettyChannelHandlerContext</code>	<code>org.jboss.netty.channel.ChannelHandlerContext</code>	ChannelHandler instance associated with the connection managed by netty.
<code>NettyConstants.NETTY_MESSAGE_EVENT</code> / <code>CamelNettyMessageEvent</code>	<code>org.jboss.netty.channel.MessageEvent</code>	MessageEvent associated with connection received by netty.
<code>NettyConstants.NETTY_REMOTE_ADDRESS</code> / <code>CamelNettyRemoteAddress</code>	<code>java.net.SocketAddress</code>	Remote address of incoming socket connection.
<code>NettyConstants.NETTY_LOCAL_ADDRESS</code> / <code>CamelNettyLocalAddress</code>	<code>java.net.SocketAddress</code>	Local address of incoming socket connection.

Usage Samples

A UDP Netty endpoint using Request-Reply and serialized object payload

```
RouteBuilder builder = new RouteBuilder() { public void configure() { from("netty:udp://localhost:5155?sync=true") .process(new Processor() { public void process(Exchange exchange) throws Exception { Poetry poetry = (Poetry) exchange.getIn().getBody();
```

```
poetry.setPoet("Dr. Sarojini Naidu"); exchange.getOut().setBody(poetry); } } }};
```

A TCP based Netty consumer endpoint using One-way communication

```
RouteBuilder builder = new RouteBuilder() { public void configure() { from("netty:tcp://localhost:5150") .to("mock:result"); } };
```

An SSL/TCP based Netty consumer endpoint using Request-Reply communication

Using the JSSE Configuration Utility

As of Camel 2.9, the Netty component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the Netty component.

Programmatic configuration of the component

```
KeyStoreParameters ksp = new KeyStoreParameters(); ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword"); KeyManagersParameters kmp = new KeyManagersParameters(); kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword"); SSLContextParameters scp = new SSLContextParameters(); scp.setKeyManagers(kmp);
NettyComponent nettyComponent = getContext().getComponent("netty", NettyComponent.class);
nettyComponent.setSSLContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
xml... <camel:sslContextParameters id="sslContextParameters"> <camel:keyManagers keyPassword="keyPassword">
<camel:keyStore resource="/users/home/server/keystore.jks" password="keystorePassword"/> </camel:keyManagers>
</camel:sslContextParameters>... ...<to uri="netty:tcp://localhost:5150?
sync=true&ssl=true&sslContextParameters=#sslContextParameters"/> ...
```

Using Basic SSL/TLS configuration on the Jetty Component

```
JndiRegistry registry = new JndiRegistry(createJndiContext()); registry.bind("password", "changeit"); registry.bind("ksf", new
File("src/test/resources/keystore.jks")); registry.bind("tsf", new File("src/test/resources/keystore.jks"));
context.createRegistry(registry); context.addRoutes(new RouteBuilder() { public void configure() { String netty_ssl_endpoint =
"netty:tcp://localhost:5150?sync=true&ssl=true&passphrase=#password" + "&keyStoreFile=#ksf&trustStoreFile=#tsf"; String
return_string = "When You Go Home, Tell Them Of Us And Say," + "For Your Tomorrow, We Give Our Today.";
from(netty_ssl_endpoint) .process(new Processor() { public void process(Exchange exchange) throws Exception {
exchange.getOut().setBody(return_string); } } );}};
```

Getting access to SSLSession and the client certificate

Available as of Camel 2.12

You can get access to the javax.net.ssl.SSLSession if you eg need to get details about the client certificate. When ssl=true then the Netty component will store the SSLSession as a header on the Camel Message as shown below:

```
SSLSession session = exchange.getIn().getHeader(NettyConstants.NETTY_SSL_SESSION, SSLSession.class); // get the first
certificate which is client certificate javax.security.cert.X509Certificate cert = session.getPeerCertificateChain()[0]; Principal
principal = cert.getSubjectDN();
```

Remember to set needClientAuth=true to authenticate the client, otherwise SSLSession cannot access information about the client certificate, and you may get an exception javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated. You may also get this exception if the client certificate is expired or not valid etc.

The option sslClientCertHeaders can be set to true which then enriches the Camel Message with headers having details about the client certificate. For example the subject name is readily available in the header

```
CamelNettySSLCertSubjectName.
```

Using Multiple Codecs

In certain cases it may be necessary to add chains of encoders and decoders to the netty pipeline. To add multiple codecs to a camel netty endpoint the 'encoders' and 'decoders' uri parameters should be used. Like the 'encoder' and 'decoder' parameters they are used to supply references (to lists of ChannelUpstreamHandlers and ChannelDownstreamHandlers) that should be added to the pipeline. Note that if encoders is specified then the encoder param will be ignored, similarly for decoders and the decoder param.

Read further above about using non shareable encoders/decoders.

The lists of codecs need to be added to the Camel's registry so they can be resolved when the endpoint is created.

```
{snippet:id=registry-beans|lang=java|url=camel/trunk/components/camel-
netty/src/test/java/org/apache/camel/component/netty/MultipleCodecsTest.java}Spring's native collections support can be used to
specify the codec lists in an application context:{snippet:id=registry-beans|lang=xml|url=camel/trunk/components/camel-
netty/src/test/resources/org/apache/camel/component/netty/multiple-codecs.xml}The bean names can then be used in netty
endpoint definitions either as a comma separated list or contained in a List e.g.
```

```
{snippet:id=routes|lang=java|url=camel/trunk/components/camel-
netty/src/test/java/org/apache/camel/component/netty/MultipleCodecsTest.java}or via spring.
```

```
{snippet:id=routes|lang=xml|url=camel/trunk/components/camel-netty/src/test/resources/org/apache/camel/component/netty/multiple-codecs.xml}
```

Closing Channel When Complete

When acting as a server you sometimes want to close the channel when, for example, a client conversion is finished. You can do this by simply setting the endpoint option `disconnect=true`.

However you can also instruct Camel on a per message basis as follows.

To instruct Camel to close the channel, you should add a header with the key `camelNettyCloseChannelWhenComplete` set to a boolean true value.

For instance, the example below will close the channel after it has written the bye message back to the client:

```
from("netty:tcp://localhost:8080").process(new Processor() { public void process(Exchange exchange) throws Exception { String body = exchange.getIn().getBody(String.class); exchange.getOut().setBody("Bye " + body); // some condition which determines if we should close if (close) { exchange.getOut().setHeader(NettyConstants.NETTY_CLOSE_CHANNEL_WHEN_COMPLETE, true); } } });
```

Adding custom channel pipeline factories to gain complete control over a created pipeline

Available as of Camel 2.5

Custom channel pipelines provide complete control to the user over the handler/interceptor chain by inserting custom handler(s), encoder(s) & decoders without having to specify them in the Netty Endpoint URL in a very simple way.

In order to add a custom pipeline, a custom channel pipeline factory must be created and registered with the context via the context registry (JNDIRegistry or the camel-spring ApplicationContextRegistry etc).

A custom pipeline factory must be constructed as follows

- A Producer linked channel pipeline factory must extend the abstract class `ClientPipelineFactory`.
- A Consumer linked channel pipeline factory must extend the abstract class `ServerPipelineFactory`.
- The classes should override the `getPipeline()` method in order to insert custom handler(s), encoder(s) and decoder(s). Not overriding the `getPipeline()` method creates a pipeline with no handlers, encoders or decoders wired to the pipeline.

The example below shows how ServerChannel Pipeline factory may be created

```
Using custom pipeline factory
public class SampleServerChannelPipelineFactory extends ServerPipelineFactory { private int maxLineSize = 1024; public ChannelPipeline getPipeline() throws Exception { ChannelPipeline channelPipeline = Channels.pipeline(); channelPipeline.addLast("encoder-SD", new StringEncoder(CharsetUtil.UTF_8)); channelPipeline.addLast("decoder-DELIM", new DelimiterBasedFrameDecoder(maxLineSize, true, Delimiters.lineDelimiter())); channelPipeline.addLast("decoder-SD", new StringDecoder(CharsetUtil.UTF_8)); // here we add the default Camel ServerChannelHandler for the consumer, to allow Camel to route the message etc. channelPipeline.addLast("handler", new ServerChannelHandler(consumer)); return channelPipeline; } }
```

The custom channel pipeline factory can then be added to the registry and instantiated/utilized on a camel route in the following way

```
Registry registry = camelContext.getRegistry(); serverPipelineFactory = new TestServerChannelPipelineFactory();
registry.bind("spf", serverPipelineFactory); context.addRoutes(news RouteBuilder() { public void configure() { String netty_ssl_endpoint = "netty:tcp://localhost:5150?serverPipelineFactory=#spf" String return_string = "When You Go Home, Tell Them Of Us And Say," + "For Your Tomorrow, We Gave Our Today."; from(netty_ssl_endpoint).process(new Processor() { public void process(Exchange exchange) throws Exception { exchange.getOut().setBody(return_string); } } ); } });
```

Reusing Netty boss and worker thread pools

Available as of Camel 2.12

Netty has two kind of thread pools: boss and worker. By default each Netty consumer and producer has their private thread pools. If you want to reuse these thread pools among multiple consumers or producers then the thread pools must be created and enlisted in the Registry.

For example using Spring XML we can create a shared worker thread pool using the `NettyWorkerPoolBuilder` with 2 worker threads as shown below:

```
xml <!-- use the worker pool builder to create to help create the shared thread pool --> <bean id="poolBuilder" class="org.apache.camel.component.netty.NettyWorkerPoolBuilder"> <property name="workerCount" value="2"/> </bean> <!-- the shared worker thread pool --> <bean id="sharedPool" class="org.jboss.netty.channel.socket.nio.WorkerPool" factory-bean="poolBuilder" factory-method="build" destroy-method="shutdown"> </bean>
```

For boss thread pool there is a `org.apache.camel.component.netty.NettyServerBossPoolBuilder` builder for Netty consumers, and a `org.apache.camel.component.netty.NettyClientBossPoolBuilder` for the Netty produces.

Then in the Camel routes we can refer to this worker pools by configuring the `workerPool` option in the URI as shown below:

```
xml <route> <from uri="netty:tcp://localhost:5021?textline=true&sync=true&workerPool=#sharedPool&orderedThreadPoolExecutor=false"/> <to uri="log:result"/> ... </route>
```

And if we have another route we can refer to the shared worker pool:

```
xml <route> <from uri="netty:tcp://localhost:5022?textline=true&sync=true&workerPool=#sharedPool&orderedThreadPoolExecutor=false"/> <to uri="log:result"/> ... </route>
```

... and so forth.

Endpoint See Also

- Netty HTTP
- MINA

NMR COMPONENT

The **nmr** component is an adapter to the Normalized Message Router (NMR) in ServiceMix, which is intended for use by Camel applications deployed directly into the OSGi container. You can exchange objects with NMR and not only XML like this is the case with the JBI specification. The interest of this component is that you can interconnect camel routes deployed in different OSGi bundles.

By contrast, the JBI component is intended for use by Camel applications deployed into the ServiceMix JBI container.

Installing in Apache Servicemix

The NMR component is provided with Apache ServiceMix. It is **not** distributed with Camel. To install the NMR component in ServiceMix, enter the following command in the ServiceMix console window:

```
features:install nmr camel-nmr
```

Installing in plain Apache Karaf

In plain Karaf the nmr component can also be installed using the servicemix artifacts:

```
features:chooseurl camel <version>
features:addurl mvn:org.apache.servicemix.nmr/apache-servicemix-nmr/1.5.0/xml/features
features:install camel-blueprint nmr camel-nmr
install -s mvn:org.apache.servicemix.camel/org.apache.servicemix.camel.component/4.4.2
```

Configuration

You also need to instantiate the NMR component. You can do this by editing your Spring configuration file, META-INF/spring/*.xml, and adding the following bean instance:

```
<beans xmlns:osgi="http://www.springframework.org/schema/osgi" ... >
  ...
  <bean id="nmr" class="org.apache.servicemix.camel.nmr.ServiceMixComponent">
    <property name="nmr">
      <osgi:reference interface="org.apache.servicemix.nmr.api.NMR" />
    </property>
  ...
</beans>
```

NMR consumer and producer endpoints

The following code:

```
from("nmr:MyServiceEndpoint")
```

Automatically exposes a new endpoint to the bus with endpoint name MyServiceEndpoint (see URI-format).

When an NMR endpoint appears at the end of a route, for example:

```
to("nmr:MyServiceEndpoint")
```

The messages sent by this producer endpoint are sent to the already deployed NMR endpoint.

URI format

```
nmr:endpointName
```

URI Options

Option	Default Value	Description
runAsSubject	false	Apache ServiceMix 4.4: When this is set to true on a consumer endpoint, the endpoint will be invoked on behalf of the subject that is set on the Exchange (i.e. the call to <code>setSubject()</code> on the <code>Message</code> object). This is useful for security reasons.
synchronous	false	Apache ServiceMix 4.4: If this is set to true on a consumer endpoint, an incoming, synchronous NMR Exchange will be handled on the sender's thread instead of being handled on a new thread of the NMR endpoint's thread pool.

<code>timeout</code>	<code>0</code>	Apache ServiceMix 4.4: When this is set to a value greater than 0, the producer endpoint will timeout if it doesn't receive a response from the NMR within the given timeout period (in milliseconds). Configuring a timeout value will switch to using synchronous interactions with the NMR instead of the usual asynchronous messaging.
<code>throwsExceptionOnFailure</code>	<code>true</code>	Apache ServiceMix 4.5.2: When this is set to <code>false</code> then NMR's exceptions (like <code>TimeoutException</code>) will be consumed silently.
<code>waitForResponse</code>	<code>true</code>	Apache ServiceMix 4.5.2: When this is set to <code>true</code> then NMR's responses will be consumed synchronously.
<code>serviceName</code>	<code>null</code>	Apache ServiceMix 4.5.3: When specify this as a QName then it could be considered when NMR looking for the target NMR endpoint during dispatch.

Examples

Consumer

```
from("nmr:MyServiceEndpoint") // consume nmr exchanges asynchronously
from("nmr:MyServiceEndpoint?synchronous=true").to() // consume nmr exchanges synchronously and use the same thread as defined by NMR ThreadPoc
```

Producer

```
from(...to("nmr:MyServiceEndpoint") // produce nmr exchanges asynchronously
from(...to("nmr:MyServiceEndpoint?timeout=10000") // produce nmr exchanges synchronously and wait till 10s to receive response
```

Using Stream bodies

If you are using a stream type as the message body, you should be aware that a stream is only capable of being read once. So if you enable DEBUG logging, the body is usually logged and thus read. To deal with this, Camel has a `streamCaching` option that can cache the stream, enabling you to read it multiple times.

```
from("nmr:MyEndpoint").streamCaching().to("xslt:transform.xsl", "bean:doSomething");
```

The stream caching is default enabled, so it is not necessary to set the `streamCaching()` option.

We store big input streams (by default, over 64K) in a temp file using `CachedOutputStream`. When you close the input stream, the temp file will be deleted.

Testing

NMR camel routes can be tested using the camel unit test approach even if they will be deployed next in different bundles on an OSGI runtime. With this aim in view, you will extend the ServiceMixNMR Mock class `org.apache.servicemix.camel.nmr.AbstractComponentTest` which will create a NMR bus, register the Camel NMR Component and the endpoints defined into the Camel routes.

```
public class ExchangeUsingNMRTest extends AbstractComponentTest {
    @Test
    public void testProcessing() throws InterruptedException {
        MockEndpoint mock = getMockEndpoint("mock:simple");
        mock.expectedBodiesReceived("Simple message body");

        template.sendBody("direct:simple", "Simple message body");

        assertMockEndpointsSatisfied();
    }

    @Override
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                from("direct:simple").to("nmr:simple");
                from("nmr:simple?synchronous=true").to("mock:simple");
            }
        };
    }
}
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

QUARTZ COMPONENT

The `quartz`: component provides a scheduled delivery of messages using the Quartz Scheduler 1.x . Each endpoint represents a different timer (in Quartz terms, a Trigger and JobDetail).

If you are using Quartz 2.x then from Camel 2.12 onwards there is a Quartz2 component you should use. Maven users will need to add the following dependency to their pom.xml for this component:

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

URI format

```
quartz://timerName?options
quartz://groupName/timerName?options
quartz://groupName/timerName?cron=expression
quartz://timerName?cron=expression
```

The component uses either a `CronTrigger` or a `SimpleTrigger`. If no cron expression is provided, the component uses a simple trigger. If no groupName is provided, the quartz component uses the Camel group name.

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

Options

Parameter	Default	Description
<code>cron</code>	<code>None</code>	Specifies a cron expression (not compatible with the <code>trigger.repeatCount</code> or <code>job.options</code>)
<code>trigger.repeatCount</code>	<code>0</code>	<code>SimpleTrigger</code> : How many times should the timer repeat?
<code>trigger.repeatInterval</code>	<code>0</code>	<code>SimpleTrigger</code> : The amount of time in milliseconds between repeated triggers.
<code>job.name</code>	<code>null</code>	Sets the job name.
<code>job.XXX</code>	<code>null</code>	Sets the job option with the <code>XXX</code> setter name.
<code>trigger.XXX</code>	<code>null</code>	Sets the trigger option with the <code>XXX</code> setter name.
<code>stateful</code>	<code>false</code>	Used for Quartz 2.0.0+ to indicate stateful job.
<code>fireAndForget</code>	<code>false</code>	New to Camel 2.2.0, if it is true will fire and forget when using Spring JMX.
<code>deleteJob</code>	<code>true</code>	Camel 2.12: If set to true, then the trigger automatically delete when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse previous trigger configuration just by changing the name. Camel 2.12: If set to true, then the trigger automatically delete when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse previous trigger configuration just by changing the name.
<code>pauseJob</code>	<code>false</code>	Camel 2.12: If set to true, then the trigger automatically pause when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse previous trigger configuration just by changing the name.
<code>usingFixedCamelContextName</code>	<code>false</code>	Camel 2.16: If it is true, JobDataMap uses the CamelContext name directly to reference the camel context. If it is false, JobDataMap uses the CamelContext management name which could be changed during the deploy time.

For example, the following routing rule will fire two timer events to the mock:`results` endpoint:

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

When using a `StatefulJob`, the `JobDataMap` is re-persisted after every execution of the job, thus preserving state for the next execution.

Running in OSGi and having multiple bundles with quartz routes

If you run in OSGi such as Apache ServiceMix, or Apache Karaf, and have multiple bundles with Camel routes that start from Quartz endpoints, then make sure if you assign

an id to the `<camelContext>` that this id is unique, as this is required by the `QuartzScheduler` in the OSGi container. If you do not set any id on `<camelContext>` then a unique id is auto assigned, and there is no problem.

Configuring quartz.properties file

By default Quartz will look for a `quartz.properties` file in the `org/quartz` directory of the classpath. If you are using WAR deployments this means just drop the `quartz.properties` in `WEB-INF/classes/org/quartz`.

However the Camel Quartz component also allows you to configure properties:

Parameter	Default	Type	Description
<code>properties</code>	<code>null</code>	<code>Properties</code>	Camel 2.4: You can configure a <code>java.util.Properties</code> instance
<code>propertiesFile</code>	<code>null</code>	<code>String</code>	Camel 2.4: File name of the properties to load from the classpath

To do this you can configure this in Spring XML as follows

```
<bean id="quartz" class="org.apache.camel.component.quartz.QuartzComponent">
  <property name="propertiesFile" value="com/mycompany/myquartz.properties"/>
</bean>
```

Enabling Quartz scheduler in JMX

You need to configure the quartz scheduler properties to enable JMX.

That is typically setting the option "`org.quartz.scheduler.jmx.export`" to a true value in the configuration file.

From Camel 2.13 onwards Camel will automatic set this option to true, unless explicit disabled.

Starting the Quartz scheduler

Available as of Camel 2.4

The Quartz component offers an option to let the Quartz scheduler be started delayed, or not auto started at all.

Parameter	Default	Type	Description
-----------	---------	------	-------------

<code>startDelayedSeconds</code>	0	int	Camel 2.4: Seconds to wait before starting the quartz scheduler.
<code>startDelayedEnabled</code>	true	boolean	Camel 2.4: Whether to start the quartz scheduler.

To do this you can configure this in Spring XML as follows

```
<bean id="quartz" class="org.apache.camel.component.quartz.QuartzComponent">
    <property name="startDelayedSeconds" value="5"/>
</bean>
```

Clustering

Available as of Camel 2.4

If you use Quartz in clustered mode, e.g. the Jobstore is clustered. Then from Camel 2.4 onwards the Quartz component will **not** pause/remove triggers when a node is being stopped/shutdown. This allows the trigger to keep running on the other nodes in the cluster.

Note: When running in clustered mode no checking is done to ensure unique job name/group for endpoints.

Message Headers

Camel adds the getters from the Quartz Execution Context as header values. The following headers are added:

`calendar`, `fireTime`, `jobDetail`, `jobInstance`, `jobRunTime`, `mergedJobDataMap`, `nextFireTime`, `previousFireTime`, `refireCount`, `result`, `scheduledFireTime`, `scheduler`, `trigger`, `triggerName`, `triggerGroup`.

The `fireTime` header contains the `java.util.Date` of when the exchange was fired.

Using Cron Triggers

Quartz supports Cron-like expressions for specifying timers in a handy format. You can use these expressions in the cron URI parameter; though to preserve valid URI encoding we allow + to be used instead of spaces. Quartz provides a little tutorial on how to use cron expressions.

For example, the following will fire a message every five minutes starting at 12pm (noon) to 6pm on weekdays:

```
from("quartz://myGroup/myTimerName?cron=0+0/5+12-18+?+*+MON-FRI").to("activemq:Totally_Rocks");
```

which is equivalent to using the cron expression

```
0 0/5 12-18 ? * MON-FRI
```

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

URI Character	Cron character
*	Space

Specifying time zone

Available as of Camel 2.8.1

The Quartz Scheduler allows you to configure time zone per trigger. For example to use a timezone of your country, then you can do as follows:

```
quartz://groupName/triggerName?cron=0+0/5+12-18+?+*+MON-FRI&trigger.timeZone=Europe/Stockholm
```

The `timeZone` value is the values accepted by `java.util.timezone`.

In Camel 2.8.0 or older versions you would have to provide your custom `String` to `java.util.Timezone` Type Converter to be able to configure this from the endpoint uri.

From Camel 2.8.1 onwards we have included such a Type Converter in the camel-core.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Quartz2
- Timer

QUICKFIX/J COMPONENT

The `quickfix` component adapts the QuickFIX/J FIX engine for using in Camel . This component uses the standard Financial Interchange (FIX) protocol for message transport.

Previous Versions

The **quickfix** component was rewritten for Camel 2.5. For information about using the **quickfix** component prior to 2.5 see the documentation section below.

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

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

URI format

`quickfix:configFile[?sessionID=sessionID&lazyCreateEngine=true|false]`

The **configFile** is the name of the QuickFIX/J configuration to use for the FIX engine (located as a resource found in your classpath). The optional **sessionID** identifies a specific FIX session. The format of the sessionID is:

(BeginString)(SenderCompID)[/(SenderSubID)[/(SenderLocationID)]]->(TargetCompID)[/(TargetSubID)[/(TargetLocationID)]]

The optional **lazyCreateEngine** (Camel 2.12.3+) parameter allows to create QuickFIX/J engine on demand. Value **true** means the engine is started when first message is send or there's consumer configured in route definition. When **false** value is used, the engine is started at the endpoint creation. When this parameter is missing, the value of component's property **lazyCreateEngines** is being used.

Example URLs:

```
quickfix:config.cfg quickfix:config.cfg?sessionID=FIX.4.2:MyTradingCompany->SomeExchange quickfix:config.cfg?
sessionID=FIX.4.2:MyTradingCompany->SomeExchange&lazyCreateEngine=true
```

ENDPOINTS

FIX sessions are endpoints for the **quickfix** component. An endpoint URI may specify a single session or all sessions managed by a specific QuickFIX/J engine. Typical applications will use only one FIX engine but advanced users may create multiple FIX engines by referencing different configuration files in **quickfix** component endpoint URLs.

When a consumer does not include a session ID in the endpoint URI, it will receive exchanges for all sessions managed by the FIX engine associated with the configuration file specified in the URI. If a producer does not specify a session in the endpoint URI then it must include the session-related fields in the FIX message being sent. If a session is specified in the URI then the component will automatically inject the session-related fields into the FIX message.

Exchange Format

The exchange headers include information to help with exchange filtering, routing and other processing. The following headers are available:

confluenceTableSmall

Header Name	Description
EventCategory	One of AppMessageReceived, AppMessageSent, AdminMessageReceived, AdminMessagesent, SessionCreated, SessionLogon, SessionLogoff. See the <code>QuickfixjEventCategory</code> enum.
SessionID	The FIX message SessionID
MessageType	The FIX MsgType tag value
DataDictionary	Specifies a data dictionary to use for parsing an incoming message. Can be an instance of a data dictionary or a resource path for a QuickFIX/J data dictionary file

The DataDictionary header is useful if string messages are being received and need to be parsed in a route. QuickFIX/J requires a data dictionary to parse certain types of messages (with repeating groups, for example). By injecting a DataDictionary header in the route after receiving a message string, the FIX engine can properly parse the data.

QuickFIX/J Configuration Extensions

When using QuickFIX/J directly, one typically writes code to create instances of logging adapters, message stores and communication connectors. The **quickfix** component will automatically create instances of these classes based on information in the configuration file. It also provides defaults for many of the common required settings and adds additional capabilities (like the ability to activate JMX support).

The following sections describe how the **quickfix** component processes the QuickFIX/J configuration. For comprehensive information about QuickFIX/J configuration, see the QFJ user manual.

Communication Connectors

When the component detects an initiator or acceptor session setting in the QuickFIX/J configuration file it will automatically create the corresponding initiator and/or acceptor connector. These settings can be in the default or in a specific session section of the configuration file.

confluenceTableSmall

Session Setting	Component Action
-----------------	------------------

`ConnectionType=initiator` Create an initiator connector

`ConnectionType=acceptor` Create an acceptor connector

The threading model for the QuickFIX/J session connectors can also be specified. These settings affect all sessions in the configuration file and must be placed in the settings default section.

confluenceTableSmall

Default/Global Setting	Component Action
<code>ThreadMode1=ThreadPerConnector</code>	Use <code>SocketInitiator</code> or <code>SocketAcceptor</code> (default)
<code>ThreadMode1=ThreadPerSession</code>	Use <code>ThreadedSocketInitiator</code> or <code>ThreadedSocketAcceptor</code>

Logging

The QuickFIX/J logger implementation can be specified by including the following settings in the default section of the configuration file. The `ScreenLog` is the default if none of the following settings are present in the configuration. It's an error to include settings that imply more than one log implementation. The log factory implementation can also be set directly on the Quickfix component. This will override any related values in the QuickFIX/J settings file.

confluenceTableSmall

Default/Global Setting	Component Action
<code>ScreenLogShowEvents</code>	Use a <code>ScreenLog</code>
<code>ScreenLogShowIncoming</code>	Use a <code>ScreenLog</code>
<code>ScreenLogShowOutgoing</code>	Use a <code>ScreenLog</code>
<code>SLF4J*</code>	Camel 2.6+. Use a <code>SLF4JLog</code> . Any of the SLF4J settings will cause this log to be used.
<code>FileLogPath</code>	Use a <code>FileLog</code>
<code>JdbcDriver</code>	Use a <code>JdbcLog</code>

Message Store

The QuickFIX/J message store implementation can be specified by including the following settings in the default section of the configuration file. The `MemoryStore` is the default if none of the following settings are present in the configuration. It's an error to include settings that imply more than one message store implementation. The message store factory implementation can also be set directly on the Quickfix component. This will override any related values in the QuickFIX/J settings file.

confluenceTableSmall

Default/Global Setting	Component Action
<code>JdbcDriver</code>	Use a <code>JdbcStore</code>
<code>FilestorePath</code>	Use a <code>FileStore</code>
<code>SleepycatDatabaseDir</code>	Use a <code>SleepycatStore</code>

Message Factory

A message factory is used to construct domain objects from raw FIX messages. The default message factory is `DefaultMessageFactory`. However, advanced applications may require a custom message factory. This can be set on the QuickFIX/J component.

JMX

confluenceTableSmall

Default/Global Setting	Component Action
<code>UseJmx</code>	if Y, then enable QuickFIX/J JMX

Other Defaults

The component provides some default settings for what are normally required settings in QuickFIX/J configuration files. `SessionStartTime` and `SessionEndTime` default to "00:00:00", meaning the session will not be automatically started and stopped. The `HeartbeatInt` (heartbeat interval) defaults to 30 seconds.

Minimal Initiator Configuration Example

[SESSION] `ConnectionType=initiator BeginString=FIX.4.4 SenderCompID=YOUR_SENDER TargetCompID=YOUR_TARGET`

Using the InOut Message Exchange Pattern

Camel 2.8+

Although the FIX protocol is event-driven and asynchronous, there are specific pairs of messages that represent a request-reply message exchange. To use an InOut exchange pattern, there should be a single request message and single reply message to the request. Examples include an OrderStatusRequest message and UserRequest.

Implementing InOut Exchanges for Consumers

Add "exchangePattern=InOut" to the QuickFIX/J endpoint URI. The `MessageOrderStatusService` in the example below is a bean with a synchronous service method. The method returns the response to the request (an `ExecutionReport` in this case) which is then sent back to the requestor session.

```
from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:MARKET->TRADER&exchangePattern=InOut")
.filter(header(QuickfixEndpoint.MESSAGE_TYPE_KEY).isEqualTo(MsgType.ORDER_STATUS_REQUEST)).bean(new
MarketOrderStatusService());
```

Implementing InOut Exchanges for Producers

For producers, sending a message will block until a reply is received or a timeout occurs. There is no standard way to correlate reply messages in FIX. Therefore, a correlation criteria must be defined for each type of InOut exchange. The correlation criteria and timeout can be specified using Exchange properties.

Description	Key String	Key Constant	Default
Correlation Criteria	"CorrelationCriteria"	QuickfixProducer.CORRELATION_CRITERIA_KEY	None
Correlation Timeout in Milliseconds	"CorrelationTimeout"	QuickfixProducer.CORRELATION_TIMEOUT_KEY	1000

The correlation criteria is defined with a `MessagePredicate` object. The following example will treat a FIX `ExecutionReport` from the specified session where the transaction type is `STATUS` and the Order ID matches our request. The session ID should be for the *requestor*, the sender and target `ComplID` fields will be reversed when looking for the reply.

```
exchange.setProperty(QuickfixProducer.CORRELATION_CRITERIA_KEY, new MessagePredicate(new SessionID(sessionID),
MsgType.EXECUTION_REPORT).withField(ExecTransType.FIELD, Integer.toString(ExecTransType.STATUS))
.withField(OrderID.FIELD, request.getString(OrderID.FIELD)));
```

Example

The source code contains an example called `RequestReplyExample` that demonstrates the InOut exchanges for a consumer and producer. This example creates a simple HTTP server endpoint that accepts order status requests. The HTTP request is converted to a FIX `OrderStatusRequestMessage`, is augmented with a correlation criteria, and is then routed to a quickfix endpoint. The response is then converted to a JSON-formatted string and sent back to the HTTP server endpoint to be provided as the web response.

The Spring configuration have changed from Camel 2.9 onwards. See further below for example.

Spring Configuration

Camel 2.6 - 2.8.x

The QuickFIX/J component includes a Spring `FactoryBean` for configuring the session settings within a Spring context. A type converter for QuickFIX/J session ID strings is also included. The following example shows a simple configuration of an acceptor and initiator session with default settings for both sessions.

```
{snippet:id=e1|lang=xml|url=camel/branches/camel-2.8.x/components/camel-quickfix/src/test/resources/org/apache/camel/component/quickfix/QuickfixSpringTest-context.xml}
```

Camel 2.9 onwards

The QuickFIX/J component includes a `QuickfixjConfiguration` class for configuring the session settings. A type converter for QuickFIX/J session ID strings is also included. The following example shows a simple configuration of an acceptor and initiator session with default settings for both sessions.

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-quickfix/src/test/resources/org/apache/camel/component/quickfix/QuickfixSpringTest-context.xml}
```

Exception handling

QuickFIX/J behavior can be modified if certain exceptions are thrown during processing of a message. If a `RejectLogon` exception is thrown while processing an incoming logon administrative message, then the logon will be rejected.

Normally, QuickFIX/J handles the logon process automatically. However, sometimes an outgoing logon message must be modified to include credentials required by a FIX counterparty. If the FIX logon message body is modified when sending a logon message (`EventCategory=Admin`) messagesent the modified message will be sent to the counterparty. It is important that the outgoing logon message is being processed *synchronously*. If it is processed asynchronously (on another thread), the FIX engine will immediately send the unmodified outgoing message when its callback method returns.

FIX Sequence Number Management

If an application exception is thrown during *synchronous* exchange processing, this will cause QuickFIX/J to not increment incoming FIX message sequence numbers and will cause a resend of the counterparty message. This FIX protocol behavior is primarily intended to handle *transport* errors rather than application errors. There are risks associated with using this mechanism to handle application errors. The primary risk is that the message will repeatedly cause application errors each time it's re-received. A better solution is to persist the incoming message (database, JMS queue) immediately before processing it. This also allows the application to process messages asynchronously without losing messages when errors occur.

Although it's possible to send messages to a FIX session before it's logged on (the messages will be sent at logon time), it is usually a better practice to wait until the session is logged on. This eliminates the required sequence number resynchronization steps at logon. Waiting for session logon can be done by setting up a route that processes the `SessionLogon` event category and signals the application to start sending messages.

See the FIX protocol specifications and the QuickFIX/J documentation for more details about FIX sequence number management.

Route Examples

Several examples are included in the QuickFIX/J component source code (test subdirectories). One of these examples implements a trivial trade execution simulation. The example defines an application component that uses the URI scheme "trade-executor".

The following route receives messages for the trade executor session and passes application messages to the trade executor component.

```
from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:MARKET->TRADER").
filter(header(QuickfixjEndpoint.EVENT_CATEGORY_KEY).isEqualTo(QuickfixjEventCategory.AppMessageReceived)). to("trade-
executor:market");
```

The trade executor component generates messages that are routed back to the trade session. The session ID must be set in the FIX message itself since no session ID is specified in the endpoint URI.

```
from("trade-executor:market").to("quickfix:examples/inprocess.cfg");
```

The trader session consumes execution report messages from the market and processes them.

```
from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:TRADER->MARKET").
filter(header(QuickfixjEndpoint.MESSAGE_TYPE_KEY).isEqualTo(MsgType.EXECUTION_REPORT)). bean(new
```

```
MyTradeExecutionProcessor());
```

QUICKFIX/J COMPONENT PRIOR TO CAMEL 2.5

The `quickfix` component is an implementation of the QuickFIX/J engine for Java . This engine allows to connect to a FIX server which is used to exchange financial messages according to FIX protocol standard.

Note: The component can be used to send/receives messages to a FIX server.

URI format

`quickfix-server:config file quickfix-client:config file`

Where **config file** is the location (in your classpath) of the quickfix configuration file used to configure the engine at the startup.

Note: Information about parameters available for quickfix can be found on QuickFIX/J web site.

The quickfix-server endpoint must be used to receive from FIX server FIX messages and quickfix-client endpoint in the case that you want to send messages to a FIX gateway.

Exchange data format

The QuickFIX/J engine is like CXF component a messaging bus using MINA as protocol layer to create the socket connection with the FIX engine gateway.

When QuickFIX/J engine receives a message, then it create a `QuickFix.Message` instance which is next received by the camel endpoint. This object is a 'mapping object' created from a FIX message formatted initially as a collection of key value pairs data. You can use this object or you can use the method 'toString' to retrieve the original FIX message.

Note: Alternatively, you can use camel bindy dataformat to transform the FIX message into your own java POJO

When a message must be send to QuickFix, then you must create a `QuickFix.Message` instance.

Lazy creating engines

From **Camel 2.12.3** onwards, you can configure the `QuickFixComponent` to lazy create and start the engines, which then only start these on-demand. For example you can use this when you have multiple Camel applications in a cluster with master/slaves. And want the slaves to be standby.

Samples

Direction : to FIX gateway

```

xml<route> <from uri="activemq:queue:fix"/> <bean ref="fixService" method="createFixMessage"/> // bean method in charge to
transform message into a QuickFix.Message <to uri="quickfix-client:META-INF/quickfix/client.cfg"/> // Quickfix engine who will
send the FIX messages to the gateway </route>
Direction : from FIX gateway
xml<route> <from uri="quickfix-server:META-INF/quickfix/server.cfg"/> // QuickFix engine who will receive the message from FIX
gateway <bean ref="fixService" method="parseFixMessage"/> // bean method parsing the QuickFix.Message <to
uri="uri="activemq:queue:fix"/> </route>
Endpoint See Also

```

PRINTER COMPONENT

Available as of Camel 2.1

The **printer** component provides a way to direct payloads on a route to a printer. Obviously the payload has to be a formatted piece of payload in order for the component to appropriately print it. The objective is to be able to direct specific payloads as jobs to a line printer in a camel flow.

This component only supports a camel producer endpoint.

The functionality allows for the payload to be printed on a default printer, named local, remote or wirelessly linked printer using the javax printing API under the covers.

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

```

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

```

URI format

Since the URI scheme for a printer has not been standardized (the nearest thing to a standard being the IETF print standard) and therefore not uniformly applied by vendors, we have chosen "lpr" as the scheme.

```

lpr://localhost/default[?options]
lpr://remotehost:port/path/to/printer[?options]

```

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

Options

Name	Default Value	Description
mediaSize	NA LETTER	Sets the stationary as defined by enumeration names in the javax.print.attribute.standard.MediaSizeName API. The default setting is to use North American Letter sized stationary. The value's supported are: A4, A3, B4, B5, Letter, Legal, Ledger, Statement, Executive, Folio, Quarto, and Tabloid.
copies	1	Sets the number of copies to print. This option is supported by the javax.print.Copies API.
sides		Sets one sided or two sided printing based on the javax.print.attribute.standard.Sides API.
flavor	DocFlavor.BYTE_ARRAY	Sets DocFlavor based on the javax.print.DocFlavor API.
contentType	AUTOTSENSE	Sets mimeTypes supported by the javax.print.DocFlavor API.
mediaTray	AUTOTSENSE	Since Camel 2.11.x sets MediaTray supported by the javax.print.DocFlavor API.
printerPrefix	null	Since Camel 2.11.x sets the prefix name of the printer, it is useful when the printer name does not start with /hostname/printer.
sendToPrinter	true	Setting this option to false prevents sending of the print data to the printer.
orientation	portrait	Sets Orientation of the page as defined by the javax.print.attribute.standard.OrientationRequested API.

Sending Messages to a Printer

Printer Producer

Sending data to the printer is very straightforward and involves creating a producer endpoint that can be sent message exchanges on in route.

Usage Samples

Example 1: Printing text based payloads on a Default printer using letter stationary and one-sided mode

```

RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from(file://inputdir?delete=true)
            .to("lpr://localhost/default?copies=2" +
                "&flavor=DocFlavor.INPUT_STREAM&" +
                "&contentType=AUTOTSENSE" +
                "&mediaSize=NA LETTER" +
                "&sides=one-sided")
    }};

```

Example 2: Printing GIF based payloads on a Remote printer using A4 stationary and one-sided mode

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("file://inputdir/?delete=true")
            .to("lpr://remotehost/sales/saleprinter" +
                "?copies=2&sides=one-sided" +
                "&contentType=GIF&mediaSize=ISO_A4" +
                "&flavor=DocFlavor.INPUT_STREAM")
    };
}
```

Example 3: Printing JPEG based payloads on a Remote printer using Japanese Postcard stationary and one-sided mode

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("file://inputdir/?delete=true")
            .to("lpr://remotehost/sales/saleprinter" +
                "?copies=2&sides=one-sided" +
                "&contentType=JPEG" +
                "&mediaSize=JAPANESE_POSTCARD" +
                "&flavor=DocFlavor.INPUT_STREAM")
    };
}
```

PROPERTIES COMPONENT

Available from Camel 2.3

URI format

```
properties:key[?options]
```

Where key is the key for the property to be looked up.

Options

Option	Type	Default	Description
cache	boolean	true	Whether or not to cache loaded properties.
encoding	String	null	Camel 2.14.2/2.15.1: The charset to use when loading properties, such as UTF-8. The default charset is ISO-8859-1 (latin1).
fallbackToAugmentedProperty	boolean	true	Camel 2.9: If true first attempt resolution of property name augmented with propertyPrefix and propertySuffix before falling back to plain property name specified. false, only the augmented property name is resolved.
defaultFallbackEnabled	boolean	true	Camel 2.19: If false the component will not attempt to find a default for the key by looking after the colon separator.
ignoreMissingLocation	boolean	false	Camel 2.10: Whether to silently ignore if a location cannot be located, such as a properties file not found.
locations	String	null	A comma separated list of one or more locations of property files to be loaded. Property resolution will use the given property files exclusively. Any default location(s) are ignored.
prefixToken	String	{	Camel 2.9: This token is used to mark the start of a property placeholder definition.
propertyPrefix	String	null	Camel 2.9: An optional prefix that's prepended to each property name prior to resolution.
propertySuffix	String	null	Camel 2.9: An optional suffix that's appended to each property name prior to resolution.
suffixToken	String	}	Camel 2.9: This token is used to mark the end of a property placeholder definition.
systemPropertiesMode	int	2	Camel 2.16: The mode to use for whether to resolve and use system properties. 0 = never - JVM system properties are never used. 1 = fallback - JVM system properties are only used as fallback if no regular property with the key exists. 2 = always - JVM system properties are always used, otherwise the regular property will be used. Bridging Spring and Camel Property Placeholders When bridging to Spring's property placeholder using org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer the configuration on

Resolving property from Java code

You can use the method `resolvePropertyPlaceholders` on the `CamelContext` to resolve a property from any Java code.

USING PROPERTYPLACEHOLDER

Available as of Camel 2.3

Camel now provides a new `PropertiesComponent` in `camel-core` which allows you to use property placeholders when defining Camel Endpoint URLs. This works much like you would do if using Spring's `<property-placeholder>` tag. However Spring has a limitation that prevents third-party frameworks from fully leveraging Spring property placeholders.

For more details see: How do I use Spring Property Placeholder with Camel XML.

Bridging Spring and Camel Property Placeholders

From Camel 2.10: Spring's property placeholder can be bridged with Camel's. See below for more details.

The property placeholder is typically used when trying to do any of the following:

- Lookup or creating endpoints.
- Lookup of beans in the Registry.
- Additional supported in Spring XML (see below in examples).
- Using Blueprint `PropertyPlaceholder` with Camel Properties component.
- Using `@PropertyInject` to inject a property in a POJO.
- Camel 2.14.1 Using default value if a property does not exists.

- **Camel 2.14.1** Include out of the box functions, to lookup property values from OS environment variables, JVM system properties, or the service idiom.
- **Camel 2.14.1** Using custom functions, which can be plugged into the property component.

Format

The value of a Camel property can be obtained by specifying its key name within a property placeholder, using the following format: `{}{key}`.

For example:

`{}{file.uri}`

where `file.uri` is the property key.

Property placeholders can be used to specify parts, or all, of an endpoint's URI by embedding one or more placeholders in the URI's string definition.

From **Camel 2.14.1**: you can specify a default value to use if a property with the key does not exists, e.g.,

`file.url:/some/path` where the default value is the text after the colon, e.g., `/some/path`.

From **Camel 2.14.1**: do not use a colon in the property key. The colon character is used as a token separator when providing a default value.

Using PropertyResolver

Camel provides a pluggable mechanism that allows third-parties to specify their own resolver to use for the lookup of properties.

Camel provides a default implementation `org.apache.camel.component.properties.DefaultPropertiesResolver` which is capable of loading properties from the file system, classpath or Registry. To indicate which source to use the location must contain the appropriate prefix.

The list of prefixes is:

Prefix	Description
<code>ref:</code>	Lookup in the Registry.
<code>file:</code>	Load the from file system.
<code>classpath:</code>	Load from the classpath (this is also the default if no prefix is provided).
<code>blueprint:</code>	Use a specific OSGi blueprint placeholder service.

Defining Location

The `PropertiesResolver` must be configured with the location(s) to use when resolving properties. One or more locations can be given. Specifying multiple locations can be done a couple of ways: using either a single comma separated string, or an array of strings.

```
javapc.setLocation("com/mycompany/myprop.properties,com/mycompany/other.properties"); pc.setLocation(new String[] {"com/mycompany/myprop.properties", "com/mycompany/other.properties"});
```

From **Camel 2.19.0**: you can set which location can be discarded if missing by setting `optional=true`, (`false` by default).

Example:

```
javapc.setLocations("com/mycompany/override.properties;optional=true,com/mycompany/defaults.properties");
```

Using System and Environment Variables in Locations

Available as of Camel 2.7

The location now supports using placeholders for JVM system properties and OS environments variables.

Example:

```
location=file:${karaf.home}/etc/foo.properties
```

In the location above we defined a location using the file scheme using the JVM system property with key `karaf.home`.

To use an OS environment variable instead you would have to prefix with env:

```
location=file:${env:APP_HOME}/etc/foo.properties
```

Where `APP_HOME` is an OS environment variable.

You can have multiple placeholders in the same location, such as:

```
location=file:${env:APP_HOME}/etc/${prop.name}.properties
```

Using System or Environment Variables to Configure Property Prefixes and Suffixes

From **Camel 2.12.5, 2.13.3, 2.14.0**: `propertyPrefix`, `propertySuffix` configuration properties support the use of placeholders for de-referencing JVM system properties and OS environments variables.

Example:

Assume the `PropertiesComponent` is configured with the following properties file:

```
textdev.endpoint = result1 test.endpoint = result2
```

The same properties file is then referenced from a route definition:

```
javaPropertiesComponent pc = context.getComponent("properties", PropertiesComponent.class);
pc.setPropertyPrefix("${stage}."); // ... context.addRoutes(new RouteBuilder() { @Override public void configure() throws
Exception { from("direct:start").to("properties:mock:{${endpoint}}"); } });
By using the configuration options propertyPrefix it's possible to change the target endpoint simply by changing the value of
the system property stage either to dev (the message will be routed to mock:result1) or test (the message will be routed
to mock:result2).
```

Configuring in Java DSL

You have to create and register the PropertiesComponent under the name properties such as:

```
javaPropertiesComponent pc = new PropertiesComponent(); pc.setLocation("classpath:com/mycompany/myprop.properties");
context.addComponent("properties", pc);
```

Configuring in Spring XML

Spring XML offers two variations to configure. You can define a spring bean as a PropertiesComponent which resembles the way done in Java DSL. Or you can use the <propertyPlaceholder> tag.

```
xml<bean id="properties" class="org.apache.camel.component.properties.PropertiesComponent"> <property name="location"
value="classpath:com/mycompany/myprop.properties"/> </bean>
```

Using the <propertyPlaceholder> tag makes the configuration a bit more fresh such as:

```
xml<camelContext ...> <propertyPlaceholder id="properties" location="com/mycompany/myprop.properties"/> </camelContext>
```

Setting the properties location through the location tag works just fine but sometime you have a number of resources to take into account and starting from **Camel 2.19.0** you can set the properties location with a dedicated propertiesLocation:

```
xml<camelContext ...> <propertyPlaceholder id="myPropertyPlaceholder"> <propertiesLocation resolver = "classpath" path =
"com/my/company/something/my-properties-1.properties" optional = "false"/> <propertiesLocation resolver = "classpath" path =
"com/my/company/something/my-properties-2.properties" optional = "false"/> <propertiesLocation resolver = "file" path =
"${karaf.home}/etc/my-override.properties" optional = "true"/> </propertyPlaceholder> </camelContext>Specifying the cache
option in XML
```

From **Camel 2.10**: Camel supports specifying a value for the cache option both inside the Spring as well as the Blueprint XML.

Using a Properties from the Registry

Available as of Camel 2.4

For example in OSGi you may want to expose a service which returns the properties as a `java.util.Properties` object.

Then you could setup the Properties component as follows:

```
xml<propertyPlaceholder id="properties" location="ref:myProperties"/>
```

Where `myProperties` is the id to use for lookup in the OSGi registry. Notice we use the `ref:` prefix to tell Camel that it should lookup the properties for the Registry.

Examples Using Properties Component

When using property placeholders in the endpoint URIs you can either use the `properties:` component or define the placeholders directly in the URI! We will show example of both cases, starting with the former.

```
java// properties cool.end=mock:result // route from("direct:start").to("properties:{cool.end}");
```

You can also use placeholders as a part of the endpoint URI:

```
java// properties cool.foo=result // route from("direct:start").to("properties:mock:{cool.foo}");
```

In the example above the to endpoint will be resolved to `mock:result`.

You can also have properties with refer to each other such as:

```
java// properties cool.foo=result cool.concat=mock:{${cool.foo}} // route from("direct:start").to("properties:mock:{${cool.concat}}");
```

Notice how `cool.concat` refer to another property.

The `properties:` component also offers you to override and provide a location in the given URI using the `locations` option:
`javafrom("direct:start").to("properties:bar.end?locations=com/mycompany/bar.properties");`

Examples

You can also use property placeholders directly in the endpoint URIs without having to use `properties:`:

```
java// properties cool.foo=result // route from("direct:start").to("mock:{${cool.foo}}");
```

And you can use them in multiple wherever you want them:

```
java// properties cool.start=direct:cool.showid=true cool.result=result // route from("{{cool.start}}").to("log:{{cool.start}}?
showBodyType=false&showExchangeId={{cool.showid}}").to("mock:{{cool.result}}");
```

You can also your property placeholders when using ProducerTemplate for example:

```
javatemplate.sendBody("{{cool.start}}", "Hello World");
```

Example with Simple language

The Simple language now also support using property placeholders, for example in the route below:

```
java// properties cheese.quote=Camel rocks // route from("direct:start") .transform().simple("Hi ${body} do you think
${properties:cheese.quote}?");
You can also specify the location in the Simple language for example:
java// bar.properties bar.quote=Beer tastes good // route from("direct:start") .transform().simple("Hi ${body}.
${properties:com/mycompany/bar.properties:bar.quote}.");
```

Additional Property Placeholder Support in Spring XML

The property placeholders is also supported in many of the Camel Spring XML tags such as `<package>`, `<packageScan>`, `<contextScan>`, `<jmxAgent>`, `<endpoint>`, `<routeBuilder>`, `<proxy>` and the others.

Example:

```
xmlUsing property placeholders in the <jmxAgent> tag<camelContext xmlns="http://camel.apache.org/schema/spring">
<propertyPlaceholder id="properties" location="org/apache/camel/spring/jmx.properties"/> <!-- we can use propery placeholders
when we define the JMX agent --> <jmxAgent id="agent" registryPort="{{myjmx.port}}" disabled="{{myjmx.disabled}}"
usePlatformMBeanServer="{{myjmx.usePlatform}}" createConnector="true" statisticsLevel="RoutesOnly"
useHostIPAddress="true"/> <route id="foo" autoStartup="false"> <from uri="seda:start"/> <to uri="mock:result"/> </route>
</camelContext>
```

Example:

```
xmlUsing property placeholders in the attributes of <camelContext><camelContext trace="{{foo.trace}}"
xmlns="http://camel.apache.org/schema/spring"> <propertyPlaceholder id="properties"
location="org/apache/camel/spring/processor/myprop.properties"/> <template id="camelTemplate" defaultEndpoint="
{{foo.cool}}"/> <route> <from uri="direct:start"/> <setHeader headerName="{{foo.header}}> <simple>$in.body World!</simple>
</setHeader> <to uri="mock:result"/> </route> </camelContext>
```

Overriding a Property Setting Using a JVM System Property

Available as of Camel 2.5

It is possible to override a property value at runtime using a JVM System property without the need to restart the application to pick up the change. This may also be accomplished from the command line by creating a JVM System property of the same name as the property it replaces with a new value.

Example:

```
javaPropertiesComponent pc = context.getComponent("properties", PropertiesComponent.class); pc.setCache(false);
System.setProperty("cool.end", "mock:override"); System.setProperty("cool.result", "override"); context.addRoutes(new
RouteBuilder() { @Override public void configure() throws Exception { from("direct:start").to("properties:cool.end");
from("direct:foo").to("properties:mock:{{cool.result}}"); } }); context.start();
getMockEndpoint("mock:override").expectedMessageCount(2); template.sendBody("direct:start", "Hello World");
template.sendBody("direct:foo", "Hello Foo"); System.clearProperty("cool.end"); System.clearProperty("cool.result");
assertMockEndpointsSatisfied();
```

Using Property Placeholders for Any Kind of Attribute in the XML DSL

Available as of Camel 2.7

If you use OSGi Blueprint then this only works from **2.11.1** or **2.10.5** on.

Previously it was only the `xs:string` type attributes in the XML DSL that support placeholders. For example often a timeout attribute would be a `xs:int` type and thus you cannot set a string value as the placeholder key. This is now possible from Camel 2.7 on using a special placeholder namespace.

In the example below we use the `prop` prefix for the namespace `http://camel.apache.org/schema/placeholder` by which we can use the `prop` prefix in the attributes in the XML DSLs. Notice how we use that in the Multicast to indicate that the option `stopOnException` should be the value of the placeholder with the key `stop`.

```
xml-beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:prop="http://camel.apache.org/schema/placeholder" xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/camel-spring.xsd http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd"> <!-- Notice in the declaration above, we have defined the prop prefix
as the Camel placeholder namespace --> <bean id="damn" class="java.lang.IllegalArgumentException"> <constructor-arg
index="0" value="Damn"/> </bean> <camelContext xmlns="http://camel.apache.org/schema/spring"> <propertyPlaceholder
id="properties" location="classpath:org/apache/camel/component/properties/myprop.properties"
xmlns="http://camel.apache.org/schema/spring"/> <route> <from uri="direct:start"/> <!-- use prop namespace, to define a
property placeholder, which maps to option stopOnException={{stop}} --> <multicast prop:stopOnException="stop"> <to
uri="mock:a"/> <throwException ref="damn"/> <to uri="mock:b"/> </multicast> </route> </camelContext> </beans>
In our properties file we have the value defined as
stop=true
```

Using Property Placeholder in the Java DSL

Available as of Camel 2.7

Likewise we have added support for defining placeholders in the Java DSL using the new `placeholder` DSL as shown in the following equivalent example:

```
javafrom("direct:start") // use a property placeholder for the option stopOnException on the Multicast EIP // which should have the value of {{stop}} key being looked up in the properties file .multicast() .placeholder("stopOnException", "stop") .to("mock:a") .throwException(new IllegalAccessException("Damn")) .to("mock:b");
```

Using Blueprint Property Placeholder with Camel Routes

Available as of Camel 2.7

Camel supports Blueprint which also offers a property placeholder service. Camel supports convention over configuration, so all you have to do is to define the OSGi Blueprint property placeholder in the XML file as shown below:

```
xml<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0" xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd"> <!-- OSGi blueprint property placeholder --> <cm:property-placeholder id="myblueprint.placeholder" persistent-id="camel.blueprint"> <!-- list some properties as needed --> <cm:default-properties> <cm:property name="result" value="mock:result"/> </cm:default-properties> </cm:property-placeholder> <camelContext xmlns="http://camel.apache.org/schema/blueprint"> <!-- in the route we can use {{ }} placeholders which will lookup in blueprint as Camel will auto detect the OSGi blueprint property placeholder and use it --> <route> <from uri="direct:start"/> <to uri="{{result}}"/> </route> </camelContext> </blueprint>
```

By default Camel detects and uses OSGi blueprint property placeholder service. You can disable this by setting the attribute `useBlueprintPropertyResolver` to false on the `<camelContext>` definition.

About placeholder syntaxes

Notice how we can use the Camel syntax for placeholders {{ }} in the Camel route, which will lookup the value from OSGi blueprint.

The blueprint syntax for placeholders is \${}. So outside the `<camelContext>` you must use the \${} syntax. Where as inside `<camelContext>` you must use {{ }} syntax. OSGi blueprint allows you to configure the syntax, so you can actually align those if you want.

You can also explicit refer to a specific OSGi blueprint property placeholder by its id. For that you need to use the Camel's `<propertyPlaceholder>` as shown in the example below:

```
xml<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0" xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd"> <!-- OSGi blueprint property placeholder --> <cm:property-placeholder id="myblueprint.placeholder" persistent-id="camel.blueprint"> <!-- list some properties as needed --> <cm:default-properties> <cm:property name="prefix.result" value="mock:result"/> </cm:default-properties> </cm:property-placeholder> <camelContext xmlns="http://camel.apache.org/schema/blueprint"> <!-- using Camel properties component and refer to the blueprint property placeholder by its id --> <propertyPlaceholder id="properties" location="blueprint:myblueprint.placeholder" prefixToken="[" suffixToken="]" propertyPrefix="prefix." /> <!-- in the route we can use {{ }} placeholders which will lookup in blueprint --> <route> <from uri="direct:start"/> <to uri="mock:foo"/> <to uri="[[result]]"/> </route> </camelContext> </blueprint>
```

Notice how we use the `blueprint` scheme to refer to the OSGi blueprint placeholder by its id. This allows you to mix and match, for example you can also have additional schemes in the location. For example to load a file from the classpath you can do:

`location="blueprint:myblueprint.placeholder,classpath:myproperties.properties"`

Each location is separated by comma.

Overriding Blueprint Property Placeholders Outside CamelContext

Available as of Camel 2.10.4

When using Blueprint property placeholder in the Blueprint XML file, you can declare the properties directly in the XML file as shown below:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-test-blueprint/src/test/resources/org/apache/camel/test/blueprint/configadmin-outside.xml}Notice that we have a `<bean>` which refers to one of the properties. And in the Camel route we refer to the other using the {{ }} notation.

Now if you want to override these Blueprint properties from an unit test, you can do this as shown below:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-test-blueprint/src/test/java/org/apache/camel/test/blueprint/ConfigAdminOverridePropertiesOutsideCamelContextTest.java}
```

To do this we override and implement the `useOverridePropertiesWithConfigAdmin` method. We can then put the properties we want to override on the given props parameter. And the return value *must* be the persistence-id of the `<cm:property-placeholder>` tag, which you define in the blueprint XML file.

Using a .cfg or .properties File For Blueprint Property Placeholders

Available as of Camel 2.10.4

When using Blueprint property placeholder in the Blueprint XML file, you can declare the properties in a .properties or .cfg file. If you use Apache ServiceMix/Karaf then this container has a convention that it loads the properties from a file in the etc directory with the naming `etc/pid.cfg`, where pid is the persistence-id.

For example in the blueprint XML file we have the `persistence-id="stuff"`, which mean it will load the configuration file as `etc/stuff.cfg`.{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-test-blueprint/src/test/resources/org/apache/camel/test/blueprint/configadmin-loadfile.xml}Now if you want to unit test this blueprint XML file, then you can override the `loadConfigAdminConfigurationFile` and tell Camel which file to load as shown below:

```
{snippet:id=e1|lang=java|url=camel/trunk/components/camel-test}
```

blueprint/src/test/java/org/apache/camel/test/blueprint/ConfigAdminLoadConfigurationFileTest.java) Notice that this method requires to return a `String[]` with 2 values. The 1st value is the path for the configuration file to load. The second value is the persistence-id of the `<cm:property-placeholder>` tag.

The `stuff.cfg` file is just a plain properties file with the property placeholders such as:
`## this is a comment greeting=Bye`

Using a .cfg file and Overriding Properties for Blueprint Property Placeholders

You can do both as well. Here is a complete example. First we have the Blueprint XML file:

```
{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-test-blueprint/src/test/resources/org/apache/camel/test/blueprint/configadmin-loadfileoverride.xml}And in the unit test class we do as follows:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-test-blueprint/src/test/java/org/apache/camel/test/blueprint/ConfigAdminLoadConfigurationFileAndOverrideTest.java}And the etc/stuff.cfg configuration file contains:  
greeting=Bye echo=Yay destination=mock:result
```

Bridging Spring and Camel Property Placeholders

Available as of Camel 2.10

The Spring Framework does not allow third-party frameworks such as Apache Camel to seamless hook into the Spring property placeholder mechanism. However you can easily bridge Spring and Camel by declaring a Spring bean with the type `org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer`, which is a Spring `org.springframework.beans.factory.config.PropertyPlaceholderConfigurer` type.

To bridge Spring and Camel you must define a single bean as shown below:{snippet:id=e1|lang=xml|title=Bridging Spring and Camel property placeholders|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/component/properties/CamelSpringPropertyPlaceholderConfigurerTest.xml} You **must not** use the spring `<context:property-placeholder>` namespace at the same time; this is not possible.

After declaring this bean, you can define property placeholders using both the Spring style, and the Camel style within the `<camelContext>` tag as shown below:{snippet:id=e2|lang=xml|title=Using bridge property placeholders|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/component/properties/CamelSpringPropertyPlaceholderConfigurerTest.xml} Notice how the hello bean is using pure Spring property placeholders using the `${} notation`. And in the Camel routes we use the Camel placeholder notation with `{}{}`.

Clashing Spring Property Placeholders with Camels Simple Language

Take notice when using Spring bridging placeholder then the spring `${} syntax` clashes with the Simple in Camel, and therefore take care.

Example:
`xml<setHeader headerName="Exchange.FILE_NAME"> <simple>{{file.rootdir}}/${in.header.CamelFileName}</simple></setHeader>`
 clashes with Spring property placeholders, and you should use `$simple${}` to indicate using the Simple language in Camel.
`xml<setHeader headerName="Exchange.FILE_NAME"> <simple>{{file.rootdir}}$/simple${in.header.CamelFileName}</simple></setHeader>`
 An alternative is to configure the `PropertyPlaceholderConfigurer` with `ignoreUnresolvablePlaceholders` option to true.

Overriding Properties from Camel Test Kit

Available as of Camel 2.10

When Testing with Camel and using the Properties component, you may want to be able to provide the properties to be used from directly within the unit test source code. This is now possible from Camel 2.10, as the Camel test kits, e.g., `CamelTestSupport` class offers the following methods

- `useOverridePropertiesWithPropertiesComponent`
- `ignoreMissingLocationWithPropertiesComponent`

So for example in your unit test classes, you can override the `useOverridePropertiesWithPropertiesComponent` method and return a `java.util.Properties` that contains the properties which should be preferred to be used.

```
{snippet:id=e1|lang=java|title=Providing properties from within unit test source|url=camel/trunk/components/camel-test-blueprint/src/test/java/org/apache/camel/test/blueprint/ConfigAdminOverridePropertiesTest.java}This can be done from any of the Camel Test kits, such as camel-test, camel-test-spring and camel-test-blueprint.
```

The `ignoreMissingLocationWithPropertiesComponent` can be used to instruct Camel to ignore any locations which was not discoverable. For example if you run the unit test, in an environment that does not have access to the location of the properties.

Using @PropertyInject

Available as of Camel 2.12

Camel allows to inject property placeholders in POJOs using the `@PropertyInject` annotation which can be set on fields and setter methods. For example you can use that with `RouteBuilder` classes, such as shown below:

```
javapublic class MyRouteBuilder extends RouteBuilder { @PropertyInject("hello") private String greeting; @Override public void configure() throws Exception { from("direct:start").transform().constant(greeting).to("${result}"); } }
```

Notice we have annotated the greeting field with `@PropertyInject` and define it to use the key `hello`. Camel will then lookup the property with this key and inject its value, converted to a String type.

You can also use multiple placeholders and text in the key, for example we can do:

```
java@PropertyInject("Hello {{name}} how are you?") private String greeting;
```

This will lookup the placeholder with their key name.

You can also add a default value if the key does not exists, such as:

```
java@PropertyInject(value = "myTimeout", defaultValue = "5000") private int timeout;
```

Using Out of the Box Functions

Available as of Camel 2.14.1

The Properties component includes the following functions out of the box

- `env` - A function to lookup the property from OS environment variables.
- `sys` - A function to lookup the property from Java JVM system properties.
- `service` - A function to lookup the property from OS environment variables using the service naming idiom.
- `service.host` - **Camel 2.16.1:** A function to lookup the property from OS environment variables using the service naming idiom returning the hostname part only.
- `service.port` - **Camel 2.16.1:** A function to lookup the property from OS environment variables using the service naming idiom returning the port part only.

As you can see these functions is intended to make it easy to lookup values from the environment. As they are provided out of the box, they can easily be used as shown below:

```
xml<camelContext xmlns="http://camel.apache.org/schema/blueprint"> <route> <from uri="direct:start"/> <to uri="{{env:SOMENAME}}"/> <to uri="{{sys:MyJvmPropertyName}}"/> </route> </camelContext>
```

You can use default values as well, so if the property does not exists, you can define a default value as shown below, where the default value is a `log:foo` and `log:bar` value.

```
xml<camelContext xmlns="http://camel.apache.org/schema/blueprint"> <route> <from uri="direct:start"/> <to uri="{{env:SOMENAME:log:foo}}"/> <to uri="{{sys:MyJvmPropertyName:log:bar}}"/> </route> </camelContext>
```

The service function is for looking up a service which is defined using OS environment variables using the service naming idiom, to refer to a service location using `hostname : port`

- `NAME_SERVICE_HOST`
- `NAME_SERVICE_PORT`

in other words the service uses `_SERVICE_HOST` and `_SERVICE_PORT` as prefix. So if the service is named `foo`, then the OS environment variables should be set as

```
export $FOO_SERVICE_HOST=myserver export $FOO_SERVICE_PORT=8888
```

For example if the `foo` service a remote HTTP service, then we can refer to the service in the Camel endpoint URI, and use the HTTP component to make the HTTP call:

```
xml<camelContext xmlns="http://camel.apache.org/schema/blueprint"> <route> <from uri="direct:start"/> <to uri="http://{{service:FOO}}/myapp"/> </route> </camelContext>
```

And we can use default values if the service has not been defined, for example to call a service on localhost, maybe for unit testing etc:

```
xml<camelContext xmlns="http://camel.apache.org/schema/blueprint"> <route> <from uri="direct:start"/> <to uri="http://{{service:FOO:localhost:8080}}/myapp"/> </route> </camelContext>
```

Using Custom Functions

Available as of Camel 2.14.1

The Properties component allow to plugin 3rd party functions which can be used during parsing of the property placeholders.

These functions are then able to do custom logic to resolve the placeholders, such as looking up in databases, do custom computations, or whatnot. The name of the function becomes the prefix used in the placeholder. This is best illustrated in the example code below

```
xml<bean id="beerFunction" class="MyBeerFunction"/> <camelContext xmlns="http://camel.apache.org/schema/blueprint"> <propertyPlaceholder id="properties" location="none" ignoreMissingLocation="true"> <propertiesFunction ref="beerFunction"/> </propertyPlaceholder> <route> <from uri="direct:start"/> <to uri="{{beer:FOO}}"/> <to uri="{{beer:BAR}}"/> </route> </camelContext>
```

Here we have a Camel XML route where we have defined the `<propertyPlaceholder>` to use a custom function, which we refer to be the bean id - e.g., the `beerFunction`. As the beer function uses `beer` as its name, then the placeholder syntax can trigger the beer function by starting with `beer:value`.

The implementation of the function is only two methods as shown below:

```
javapublic static final class MyBeerFunction implements PropertiesFunction { @Override public String getName() { return "beer"; }  
    @Override public String apply(String remainder) { return "mock:" + remainder.toLowerCase(); } }  
The function must implement the org.apache.camel.component.properties.PropertiesFunction interface. The  
method getName is the name of the function, e.g., beer. And the apply method is where we implement the custom logic to do.  
As the sample code is from an unit test, it just returns a value to refer to a mock endpoint.
```

To register a custom function from Java code is as shown below:

```
javaPropertiesComponent pc = context.getComponent("properties", PropertiesComponent.class); pc.addFunction(new  
MyBeerFunction());
```

SEE ALSO

- Properties component

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Jasypt for using encrypted values e.g., passwords, in properties.

REF COMPONENT

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

URI format

```
ref:someName[?options]
```

Where **someName** is the name of an endpoint in the Registry (usually, but not always, the Spring registry). If you are using the Spring registry, someName would be the bean ID of an endpoint in the Spring registry.

Runtime lookup

This component can be used when you need dynamic discovery of endpoints in the Registry where you can compute the URI at runtime. Then you can look up the endpoint using the following code:

```
// lookup the endpoint  
String myEndpointRef = "bigspenderOrder";  
Endpoint endpoint = context.getEndpoint("ref:" + myEndpointRef);  
  
Producer producer = endpoint.createProducer();  
Exchange exchange = producer.createExchange();  
exchange.getIn().setBody(payloadToSend);  
// send the exchange  
producer.process(exchange);  
...
```

And you could have a list of endpoints defined in the Registry such as:

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">  
    <endpoint id="normalOrder" uri="activemq:order.slow"/>  
    <endpoint id="bigspenderOrder" uri="activemq:order.high"/>  
    ...  
</camelContext>
```

Sample

In the sample below we use the **ref:** in the URI to reference the endpoint with the spring ID, **endpoint2**:

```
<bean id="mybean" class="org.apache.camel.spring.example.DummyBean">  
    <property name="endpoint" ref="endpoint1"/>  
</bean>  
  
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">  
    <jmxAgent id="agent" disabled="true"/>  
    <endpoint id="endpoint1" uri="direct:start"/>  
    <endpoint id="endpoint2" uri="mock:end"/>  
  
    <route>  
        <from ref="endpoint1"/>  
        <to uri="ref:endpoint2"/>  
    </route>  
</camelContext>
```

You could, of course, have used the `ref` attribute instead:

```
<to ref="endpoint2"/>
```

Which is the more common way to write it.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

RESTLET COMPONENT

The **Restlet** component provides Restlet based endpoints for consuming and producing RESTful resources.

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

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

URI format

`javarestlet:restletUrl[?options]`

Format of `restletUrl`:

`javaprotocol://hostname[:port]/[resourcePattern]`

Restlet promotes decoupling of protocol and application concerns. The reference implementation of Restlet Engine supports a number of protocols. However, we have tested the HTTP protocol only. The default port is port 80. We do not automatically switch default port based on the protocol yet.

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

It seems Restlet is case sensitive in understanding headers. For example to use `content-type`, use `Content-Type`, and for location use `Location` and so on.

We have received a report about drop in performance in camel-restlet in Camel 2.14.0 and 2.14.1. We have reported this to the Restlet team in issue 996. To remedy the issue then from Camel 2.14.2 onwards you can set `synchronous=true` as option on the endpoint uris, Or set it on the `RestletComponent` as a global option so all endpoints inherit this option.

Options

Name	Default Value	Description
<code>headerFilterStrategy=#refName</code>	An instance of <code>RestletHeaderFilterstrategy</code>	Use the <code>#</code> notation (<code>headerFilterStrategy=#refName</code>) to reference a header filter strategy in the Camel Registry. The strategy will be plugged into the restlet binding if it is <code>HeaderFilterStrategyAware</code> .
<code>restletBinding=#refName</code>	An instance of <code>DefaultRestletBinding</code>	The bean ID of a <code>RestletBinding</code> object in the Camel Registry.
<code>restletMethod</code>	<code>GET</code>	On a producer endpoint, specifies the request method to use. On a consumer endpoint, specifies that the endpoint consumes only <code>restletMethod</code> requests. The string value is converted to <code>org.restlet.data.Method</code> by the <code>Method.valueOf(String)</code> method.
<code>restletMethods</code>	<code>None</code>	Consumer only Specify one or more methods separated by commas (e.g. <code>restletMethods=post,put</code>) to be serviced by a restlet consumer endpoint. If both <code>restletMethod</code> and <code>restletMethods</code> options are specified, the <code>restletMethod</code> setting is ignored.
<code>restletRealm=#refName</code>	<code>null</code>	The bean ID of the Realm Map in the Camel Registry.
<code>restletUriPatterns=#refName</code>	<code>None</code>	Consumer only Specify one ore more URI templates to be serviced by a restlet consumer endpoint, using the <code>#</code> notation to reference a

		<i>List<String></i> in the Camel Registry. If a URI pattern has been defined in the endpoint URI, both the URI pattern defined in the endpoint and the restletUriPatterns option will be honored.
throwExceptionOnFailure (2.6 or later)	true	*Producer only * Throws exception on a producer failure.
connectionTimeout	300000	Since Camel 2.12.3 Producer only The Client will give up connection if the connection is timeout, 0 for unlimited wait.
socketTimeout	300000	Since Camel 2.12.3 Producer only The Client socket receive timeout, 0 for unlimited wait.
disableStreamCache	false	Camel 2.14: Determines whether or not the raw input stream from Jetty is cached or not (Camel will read the stream into a in memory/overflow to file, Stream caching) cache. By default Camel will cache the Jetty input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultRestletBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times.
streamRepresentation	false	Camel 2.16.4/2.17.2: Producer only Whether to support stream representation as response from calling a REST service using the restlet producer. If the response is streaming then this option can be enabled to use an <code>java.io.InputStream</code> as the message body on the Camel Message body. If using this option you may want to enable the autoCloseStream option as well to ensure the input stream is closed when the Camel Exchange is done being routed. However if you need to read the stream outside a Camel route, you may need to not auto close the stream.
autoCloseStream	false	Camel 2.16.4/2.17.2: Producer only Whether to auto close the stream representation as response from calling a REST service using the restlet producer. If the response is streaming and the option streamRepresentation is enabled then you may want to auto close the <code>InputStream</code> from the streaming response to ensure the input stream is closed when the Camel Exchange is done being routed. However if you need to read the stream outside a Camel route, you may need to not auto close the stream.
cookieHandler	null	Camel 2.19: Producer only: Configure a cookie handler to maintain a HTTP session

Component Options

The Restlet component can be configured with the following options. Notice these are **component** options and cannot be configured on the endpoint, see further below for an example.

Name	Default Value	Description
controllerDaemon	true	Camel 2.10: Indicates if the controller thread should be a daemon (not blocking JVM exit).
controllersleepTimeMs	100	Camel 2.10: Time for the controller thread to sleep between each control.
inboundbuffersize	8192	Camel 2.10: The size of the buffer when reading messages.

minThreads	1	Camel 2.10: Minimum threads waiting to service requests.
maxThreads	10	Camel 2.10: Maximum threads that will service requests.
lowThreads	8	Camel 2.13: Number of worker threads determining when the connector is considered overloaded.
maxQueued	0	Camel 2.13: Maximum number of calls that can be queued if there aren't any worker thread available to service them. If the value is '0', then no queue is used and calls are rejected if no worker thread is immediately available. If the value is '-1', then an unbounded queue is used and calls are never rejected.
maxConnectionsPerHost	-1	Camel 2.10: Maximum number of concurrent connections per host (IP address).
maxTotalConnections	-1	Camel 2.10: Maximum number of concurrent connections in total.
outboundBufferSize	8192	Camel 2.10: The size of the buffer when writing messages.
persistingConnections	true	Camel 2.10: Indicates if connections should be kept alive after a call.
pipeliningConnections	false	Camel 2.10: Indicates if pipelining connections are supported.
threadMaxIdleTimeMs	60000	Camel 2.10: Time for an idle thread to wait for an operation before being collected.
useForwardedForHeader	false	Camel 2.10: Lookup the "X-Forwarded-For" header supported by popular proxies and caches and uses it to populate the Request.getClientAddresses() method result. This information is only safe for intermediary components within your local network. Other addresses could easily be changed by setting a fake header and should not be trusted for serious security checks.
reuseAddress	true	Camel 2.10.5/2.11.1: Enable/disable the SO_REUSEADDR socket option. See java.io.ServerSocket#reuseAddress property for additional details.
disableStreamCache	false	Camel 2.14: Determines whether or not the raw input stream from Jetty is cached or not (Camel will read the stream into a memory/overflow to file, Stream caching) cache. By default Camel will cache the Jetty input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultRestletBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times.
enabledConverters	null	Camel 2.18: By default, Restlet engine loads all the extension it finds at run-time and this option filter out all the converters except those explicitly listed using full qualified class name or simple class name. i.e. by setting enabledConverters=JacksonConverter, GsonConverter the RestletComponent will remove all the converters loaded by the Restlet engine except Jackson and Gson. Note that you still need to add the extensions as dependency.

Message Headers

confluenceTableSmall

Name	Type	Description
Content-Type	String	Specifies the content type, which can be set on the OUT message by the application/processor. The value is the content-type of the response message. If this header is not set, the content type is based on the object type of the OUT message body. In Camel 2.3 onward, if the Content-Type header is specified in the Camel IN message, the value of the header determines the content type for the Restlet request message. Otherwise, it is defaulted to "application/x-www-form-urlencoded". Prior to release 2.3, it is not possible to change the request content type default.
CamelAcceptContentType	String	Since Camel 2.9.3, 2.10.0: The HTTP Accept request header.
CamelHttpMethod	String	The HTTP request method. This is set in the IN message header.
CamelHttpQuery	String	The query string of the request URI. It is set on the IN message by defaultRestletBinding when the restlet component receives a request.
CamelHttpResponseCode	String or Integer	The response code can be set on the OUT message by the application/processor. The value is the response code of the response message. If this header is not set, the response code is set by the restlet runtime engine.
CamelHttpuri	String	The HTTP request URI. This is set in the IN message header.
CamelRestletLogin	String	Login name for basic authentication. It is set on the IN message by the application and gets filtered before the restlet request header by

		Camel.
CamelRestletPassword	string	Password name for basic authentication. It is set on the IN message by the application and gets filtered before the restlet request header by Camel.
CamelRestletRequest	Request	Camel 2.8: The org.restlet.Request object which holds all request details.
CamelRestletResponse	Response	Camel 2.8: The org.restlet.Response object. You can use this to create responses using the API from Restlet. See examples below.
org.restlet.*		Attributes of a Restlet message that get propagated to Camel IN headers.
cache-control	String or List<CacheDirective>	Camel 2.11: User can set the cache-control with the String value or the List of CacheDirective of Restlet from the camel message header.

Message Body

Camel will store the restlet response from the external server on the OUT body. All headers from the IN message will be copied to the OUT message, so that headers are preserved during routing.

Samples

Restlet Endpoint with Authentication

The following route starts a restlet consumer endpoint that listens for POST requests on `http://localhost:8080`. The processor creates a response that echoes the request body and the value of the `id` header.

```
{snippet:id=consumer_route|lang=java|url=camel/trunk/components/camel-restlet/src/test/java/org/apache/camel/component/restlet/route/TestRouteBuilder.java}The restletRealm setting in the URI query is used to look up a Realm Map in the registry. If this option is specified, the restlet consumer uses the information to authenticate user logins. Only authenticated requests can access the resources. In this sample, we create a Spring application context that serves as a registry. The bean ID of the Realm Map should match the restletRealmRef.
```

```
{snippet:id=realm|lang=xml|url=camel/trunk/components/camel-restlet/src/test/resources/org/apache/camel/component/restlet/camel-context.xml}The following sample starts a direct endpoint that sends requests to the server on http://localhost:8080 (that is, our restlet consumer endpoint).
```

```
{snippet:id=producer_route|lang=java|url=camel/trunk/components/camel-restlet/src/test/java/org/apache/camel/component/restlet/route/TestRouteBuilder.java}That is all we need. We are ready to send a request and try out the restlet component:{snippet:id=auth_request|lang=java|url=camel/trunk/components/camel-restlet/src/test/java/org/apache/camel/component/restlet/RestletRouteBuilderAuthTest.java}The sample client sends a request to the direct:start-auth endpoint with the following headers:
```

- `CamelRestletLogin` (used internally by Camel)
- `CamelRestletPassword` (used internally by Camel)
- `id` (application header)

Note

`org.apache.camel.restlet.auth.login` and `org.apache.camel.restlet.auth.password` will not be propagated as Restlet header.

The sample client gets a response like the following:

textreceived [<order foo='1'>] as an order id = 89531

Single restlet endpoint to service multiple methods and URI templates

It is possible to create a single route to service multiple HTTP methods using the `restletMethods` option. This snippet also shows how to retrieve the request method from the header:

```
{snippet:id=routeDefinition|lang=java|url=camel/trunk/components/camel-restlet/src/test/java/org/apache/camel/component/restlet/RestletMultiMethodsEndpointTest.java}In addition to servicing multiple methods, the next snippet shows how to create an endpoint that supports multiple URI templates using the restletUriPatterns option. The request URI is available in the header of the IN message as well. If a URI pattern has been defined in the endpoint URI (which is not the case in this sample), both the URI pattern defined in the endpoint and the restletUriPatterns option will be honored.{snippet:id=routeDefinition|lang=java|url=camel/trunk/components/camel-restlet/src/test/java/org/apache/camel/component/restlet/RestletMultiUriTemplatesEndpointTest.java}The restletUriPatterns="#uriTemplates option references the List<String> bean defined in the Spring XML configuration.
```

```
xml<util:list id="uriTemplates"> <value>/users/{username}</value> <value>/atom/collection/{id}/component/{cid}</value></util:list>
```

Using Restlet API to populate response

Available as of Camel 2.8

You may want to use the `org.restlet.Response` API to populate the response. This gives you full access to the Restlet API and fine grained control of the response. See the route snippet below where we generate the response from an inlined Camel Processor:
`{snippet:id=e1|title=Generating response using Restlet Response API|lang=java|url=camel/trunk/components/camel-restlet/src/test/java/org/apache/camel/component/restlet/RestletRequestAndResponseAPITest.java}`

Configuring max threads on component

To configure the max threads options you must do this on the component, such as:

```
xml<bean id="restlet" class="org.apache.camel.component.restlet.RestletComponent"> <property name="maxThreads" value="100"/> </bean>
```

Using the Restlet servlet within a webapp

Available as of Camel 2.8

There are three possible ways to configure a Restlet application within a servlet container and using the subclassed `SpringServerServlet` enables configuration within Camel by injecting the Restlet Component.

Use of the Restlet servlet within a servlet container enables routes to be configured with relative paths in URIs (removing the restrictions of hard-coded absolute URLs) and for the hosting servlet container to handle incoming requests (rather than have to spawn a separate server process on a new port).

To configure, add the following to your `camel-context.xml`:

```
xml<camelContext> <route id="RS_RestletDemo"> <from uri="restlet:/demo/{id}" /> <transform> <simple>Request type : ${header.CamelHttpMethod} and ID : ${header.id}</simple> </transform> </route> </camelContext> <bean id="RestletComponent" class="org.restlet.Component" /> <bean id="RestletComponentService" class="org.apache.camel.component.restlet.RestletComponent"> <constructor-arg index="0" > <ref bean="RestletComponent" /> </constructor-arg> </bean>
And add this to your web.xml;
xml<!-- Restlet Servlet --> <servlet> <servlet-name>RestletServlet</servlet-name> <servlet-class>org.restlet.ext.spring.SpringServerServlet</servlet-class> <init-param> <param-name>org.restlet.component</param-name> <param-value>RestletComponent</param-value> </init-param> </servlet> <servlet-mapping> <servlet-name>RestletServlet</servlet-name> <url-pattern>/rs/*</url-pattern> </servlet-mapping>
You will then be able to access the deployed route at http://localhost:8080/mywebapp/rs/demo/1234 where;
```

`localhost:8080` is the server and port of your servlet container

`mywebapp` is the name of your deployed webapp

Your browser will then show the following content;

"Request type : GET and ID : 1234"

You will need to add dependency on the Spring extension to restlet which you can do in your Maven `pom.xml` file:

```
xml<dependency> <groupId>org.restlet.jee</groupId> <artifactId>org.restlet.ext.spring</artifactId> <version>${restlet-version}</version> </dependency>
```

And you would need to add dependency on the restlet maven repository as well:

```
xml<repository> <id>maven-restlet</id> <name>Public online Restlet repository</name> <url>http://maven.restlet.org</url> </repository>
```

Endpoint See Also

RMI COMPONENT

The `rmi`: component binds Exchanges to the RMI protocol (JRMP).

Since this binding is just using RMI, normal RMI rules still apply regarding what methods can be invoked. This component supports only Exchanges that carry a method invocation from an interface that extends the `Remote` interface. All parameters in the method should be either `Serializable` or `Remote` objects.

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

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

URI format

```
rmi://rmi-registry-host:rmi-registry-port/registry-path[?options]
```

For example:

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

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

Options

Name	Default Value	Description
method	null	You can set the name of the method to invoke.
remoteInterfaces	null	It's now possible to use this option from Camel 2.7; in the XML DSL, it can be a list of interface names separated by commas.

Using

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

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

To bind an existing camel processor or service in an RMI registry, define an RMI endpoint as follows:

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

Note that when binding an RMI consumer endpoint, you must specify the `Remote interfaces exposed`.

In XML DSL you can do as follows from **Camel 2.7** onwards:

```
<camel:route>
  <from uri="rmi://localhost:37541/helloServiceBean?remoteInterfaces=org.apache.camel.example.osgi.HelloService"/>
  <to uri="bean:helloServiceBean"/>
</camel:route>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

RSS COMPONENT

The `rss:` component is used for polling RSS feeds. Camel will default poll the feed every 60th seconds.

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

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

Note: The component currently only supports polling (consuming) feeds.

Camel-rss internally uses a patched version of ROME hosted on ServiceMix to solve some OSGi class loading issues.

URI format

`rss:rssUri`

Where `rssuri` is the URI to the RSS feed to poll.

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

Options

confluenceTableSmall

Property	Default	Description
splitEntries	true	If true, Camel splits a feed into its individual entries and returns each entry, poll by poll. For example, if a feed contains seven entries, Camel returns the first entry on the first poll, the second entry on the second poll, and so on. When no more entries are left in the feed, Camel contacts the remote RSS URI to obtain a new feed. If false, Camel obtains a fresh feed on every poll and returns all of the feed's entries.
filter	true	Use in combination with the <code>splitEntries</code> option in order to filter returned entries. By default, Camel applies the <code>updatedateFilter</code> filter, which returns only new entries from the feed, ensuring that the consumer endpoint never receives an entry more than once. The filter orders the entries chronologically, with the newest returned last.
throttleEntries	true	Camel 2.5: Sets whether all entries identified in a single feed poll should be delivered immediately. If true, only one entry is processed per consumer.delay. Only applicable when <code>splitEntries</code> is set to true.
lastUpdate	null	Use in combination with the <code>filter</code> option to block entries earlier than a specific

		date/time (uses the entry.updated timestamp). The format is: yyyy-MM-ddTHH:mm:ss. Example: 2007-12-24T17:45:59.
feedHeader	true	Specifies whether to add the ROME SyndFeed object as a header.
sortEntries	false	If splitEntries is true, this specifies whether to sort the entries by updated date.
consumer.delay	60000	Delay in milliseconds between each poll.
consumer.initialDelay	1000	Milliseconds before polling starts.
consumer.userFixedDelay	false	Set to true to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
username		Camel 2.16: For basic authentication when polling from a HTTP feed
password		Camel 2.16: For basic authentication when polling from a HTTP feed

Exchange data types

Camel initializes the In body on the Exchange with a ROME syndFeed. Depending on the value of the splitEntries flag, Camel returns either a SyndFeed with one SyndEntry or a java.util.List of SyndEntries.

confluenceTableSmall

Option	Value	Behavior
splitEntries	true	A single entry from the current feed is set in the exchange.
splitEntries	false	The entire list of entries from the current feed is set in the exchange.

Message Headers

confluenceTableSmall

Header	Description
CamelRSSFeed	The entire SyncFeed object.

RSS Dataformat

The RSS component ships with an RSS dataformat that can be used to convert between String (as XML) and ROME RSS model objects.

- marshal = from ROME syndFeed to XML String
- unmarshal = from XML string to ROME syndFeed

A route using this would look something like this:{snippet:id=ex|lang=java|url=camel/trunk/components/camel-rss/src/test/java/org/apache/camel/dataformat/rss/RssDataFormatTest.java}The purpose of this feature is to make it possible to use Camel's lovely built-in expressions for manipulating RSS messages. As shown below, an XPath expression can be used to filter the RSS message:{snippet:id=ex|lang=java|url=camel/trunk/components/camel-rss/src/test/java/org/apache/camel/dataformat/rss/RssFilterWithXPathTest.java}

Query parameters

If the URL for the RSS feed uses query parameters, this component will understand them as well, for example if the feed uses alt=rss, then you can for example do
from("rss:http://someserver.com/feeds/posts/default?alt=rss&splitEntries=false&consumer.delay=1000").to("bean:rss");

Filtering entries

You can filter out entries quite easily using XPath, as shown in the data format section above. You can also exploit Camel's Bean Integration to implement your own conditions. For instance, a filter equivalent to the XPath example above would be:

{snippet:id=ex1|lang=java|url=camel/trunk/components/camel-rss/src/test/java/org/apache/camel/component/rss/RssFilterTest.java}The custom bean for this would be:
{snippet:id=ex2|lang=java|url=camel/trunk/components/camel-rss/src/test/java/org/apache/camel/component/rss/RssFilterTest.java}Endpoint See Also

- Atom

Unable to render {include} The included page could not be found.

SEDA COMPONENT

The seda: component provides asynchronous SEDA behavior, so that messages are exchanged on a BlockingQueue and consumers are invoked in a separate thread from the producer.

Note that queues are only visible within a *single* CamelContext. If you want to communicate across CamelContext instances (for example, communicating between Web applications), see the VM component.

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

Synchronous

The Direct component provides synchronous invocation of any consumers when a producer sends a message exchange.

URI format

`sedasomeName[?options]`

Where `someName` can be any string that uniquely identifies the endpoint within the current CamelContext.

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

Options

Name	Since	Default	Description
<code>size</code>			<p>The maximum capacity of the sedasomeName queue, i.e., the number of messages it can hold. The default value in Camel 2.2 or older is 1000. From Camel 2.3: the size is unbounded by default.</p>
<code>concurrentConsumers</code>		1	<p>Note: Care should be taken when using this option. The size is determined by the value specified <i>when the first endpoint is created</i>. Each endpoint must therefore specify the same size. From Camel 2.11: a validation is taken place to ensure if using mixed queue sizes for the same queue name, Camel would detect this and fail creating the endpoint.</p>
<code>waitForTaskToComplete</code>		<code>IfReplyExpected</code>	<p>Number of concurrent threads processing exchanges. Option to specify whether the caller should wait for the asynchronous task to complete before continuing. The following options are supported:</p> <ul style="list-style-type: none"> • Always • Never • <code>IfReplyExpected</code> <p>The first two values are self-explanatory. The last value, <code>IfReplyExpected</code>, will only wait if the message is Request Reply based. See Async messaging for more details.</p>
<code>timeout</code>		30000	<p>Timeout (in milliseconds) before a sedasomeName producer will stop waiting for an asynchronous task to complete. See <code>waitForTaskToComplete</code> and <code>Async</code> for more details. From Camel 2.2: you can now disable timeout by using 0 or a negative value.</p>
<code>multipleConsumers</code>	2.2	false	<p>Specifies whether multiple consumers are allowed. If enabled, you can use SEDA for Publish-Subscribe messaging. That is, you can send a message to the sedasomeName queue and have each consumer receive a copy of the message. When enabled, this option should be specified on every consumer endpoint.</p>
<code>limitConcurrentConsumers</code>	2.3	true	<p>Whether to limit the number of concurrent consumers to the maximum of 500. By default, an exception will be thrown if a sedasomeName endpoint is configured with a greater number. You can disable that check by turning this option off.</p>
<code>blockWhenFull</code>	2.9	false	<p>Whether a thread that sends messages to a full sedasomeName queue will block until the queue's capacity is no longer exhausted. By default, an exception will be thrown stating that the queue is full. By enabling this option, the calling thread will instead block and wait until the message can be accepted.</p>
<code>queueSize</code>	2.9		<p>Component only: the maximum size (capacity of the number of messages it can hold) of the sedasomeName queue. This option is used when <code>size</code> is not specified.</p>
<code>pollTimeout</code>	2.9.3	1000	<p>Consumer only: the timeout used when polling. When a timeout occurs, the consumer can check whether it is allowed</p>

			to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.
purgeWhenStopping	2.11.1	false	Whether to purge the task queue when stopping the consumer/route. This allows to stop faster, as any pending messages on the queue is discarded.
queue	2.12.0	null	Define the queue instance which will be used by seda endpoint
queueFactory	2.12.0	null	Define the queueFactory which could create the queue for the seda endpoint
failIfNoConsumers	2.12.0	false	Whether the producer should fail by throwing an exception when sending to a seda queue with no active consumers. Only one of the options <code>discardIfNoConsumers</code> and <code>failIfNoConsumers</code> can be enabled at the same time.
discardIfNoConsumers	2.16	false	Whether the producer should discard the message (do not add the message to the queue) when sending to a seda queue with no active consumers. Only one of the options <code>discardIfNoConsumers</code> and <code>failIfNoConsumers</code> can be enabled at the same time.

Choosing BlockingQueue implementation

Available as of Camel 2.12

```
xml<bean id="arrayQueue" class="java.util.ArrayBlockingQueue"> <constructor-arg index="0" value="10"> <!-- size -->
<constructor-arg index="1" value="true"> <!-- fairness --> </bean> <!-- ... --> <from uri="seda:array?queue=#arrayQueue"/>
By default, the seda component instantiates a LinkedBlockingQueue. However, a different implementation can be chosen by specifying a custom BlockingQueue implementation. When a custom implementation is configured the size option is ignored.
```

The list of available `BlockingQueueFactory` implementations includes:

- `LinkedBlockingQueueFactory`
- `ArrayBlockingQueueFactory`
- `PriorityBlockingQueueFactory`

```
xml<bean id="priorityQueueFactory" class="org.apache.camel.component.seda.PriorityBlockingQueueFactory"> <property
name="comparator"> <bean class="org.apache.camel.demo.MyExchangeComparator"/> </property> </bean> <!-- ...and later -->
<from uri="seda:priority?queueFactory=#priorityQueueFactory&size=100"/> <!-- ... -->
```

Use of Request Reply

The SEDA component supports using Request Reply, where the caller will wait for the Async route to complete. For instance:

```
javafrom("mina:tcp://0.0.0.0:9876?textline=true&sync=true") .to("seda:input"); from("seda:input") .to("bean:processInput")
.to("bean:createResponse");
```

In the route above, we have a TCP listener on port 9876 that accepts incoming requests. The request is routed to the `seda:input` queue. As it is a Request Reply message, we wait for the response. When the consumer on the `seda:input` queue is complete, it copies the response to the original message response.

until 2.2: Works only with 2 endpoints

Using Request Reply over SEDA or VM only works with 2 endpoints. You **cannot** chain endpoints by sending to A -> B -> C etc. Only between A -> B. The reason is the implementation logic is fairly simple. To support 3+ endpoints makes the logic much more complex to handle ordering and notification between the waiting threads properly.

This has been improved in **Camel 2.3**, which allows you to chain as many endpoints as you like.

Concurrent consumers

By default, the SEDA endpoint uses a single consumer thread, but you can configure it to use concurrent consumer threads. So, instead of thread pools you can use:

```
javafrom("seda:stageName?concurrentConsumers=5") .process(...)
```

As for the difference between the two, note a *thread pool* can increase/shrink dynamically at runtime depending on load, whereas the number of concurrent consumers is always fixed.

Thread pools

Be aware that adding a thread pool to a **seda** endpoint by doing something like:

```
javafrom("seda:stageName") .thread(5) .process(...)
```

Can wind up with two `BlockingQueues`: one from the **seda** endpoint, and one from the workqueue of the thread pool, which may not be what you want. Instead, you might wish to configure a Direct endpoint with a thread pool, which can process messages both synchronously and asynchronously. For example:

```
javafrom("direct:stageName") .thread(5) .process(...)
```

You can also directly configure number of threads that process messages on a **seda** endpoint using the `concurrentConsumers` option.

Sample

In the route below we use the SEDA queue to send the request to this asynchronous queue to be able to send a fire-and-forget message for further processing in another thread, and return a constant reply in this thread to the original caller.
`INLINE{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/seda/SedaAsyncRouteTest.java}` Here we send a Hello World message and expects the reply to be OK.
`INLINE{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/seda/SedaAsyncRouteTest.java}` The Hello world message will be consumed from the **seda** queue from another thread for further processing. Since this is from a unit test, it will be sent to a mock endpoint where we can do assertions in the unit test.

Using multipleConsumers

Available as of Camel 2.2

In this example we have defined two consumers and registered them as spring beans.
`INLINE{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/example/fooEventRoute.xml}` Since we have specified `multipleConsumers=true` on the **seda** foo endpoint we can have those two consumers receive their own copy of the message as a kind of pub-sub style messaging.

As the beans are part of an unit test they simply send the message to a mock endpoint. Note the use of `@consume` to consume from the **seda** queue.
`INLINE{snippet:id=e1|lang=java|url=camel/trunk/components/camel-spring/src/test/java/org/apache/camel/spring/example/FooEventConsumer.java}`

Extracting Queue Information.

If needed, information such as queue size, etc. can be obtained without using JMX in this fashion:

```
javaSedaEndpoint seda = context.getEndpoint("seda:xxxx");
int size = seda.getExchanges().size();
```

Endpoint See Also

- VM
- Disruptor
- Direct
- Async

SERVLET COMPONENT

The **servlet**: component provides HTTP based endpoints for consuming HTTP requests that arrive at a HTTP endpoint that is bound to a published Servlet.

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

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

Servlet is stream based, which means the input it receives is submitted to Camel as a stream. That means you will only be able to read the content of the stream **once**. If you find a situation where the message body appears to be empty or you need to access the data multiple times (eg: doing multicasting, or redelivery error handling) you should use Stream caching or convert the message body to a string which is safe to be read multiple times.

URI format

`servlet://relative_path[?options]`

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

Options

confluenceTableSmall

Name	Default Value	Description
<code>httpBindingRef</code>	<code>null</code>	Reference to an <code>org.apache.camel.component.http.HttpBinding</code> in the Registry. A <code>HttpBinding</code> implementation can be used to customize how to write a response.
<code>httpBinding</code>	<code>null</code>	Camel 2.16: Reference to an <code>org.apache.camel.component.http.HttpBinding</code> in the Registry. A <code>HttpBinding</code> implementation can be used to customize how to write a response.
<code>matchOnUriPrefix</code>	<code>false</code>	

Whether or not the `CamelServlet` should try to find a target consumer by matching the URI prefix, if no exact match is found.

<code>serviceName</code>	<code>CamelServlet</code>	Specifies the servlet name that the servlet endpoint will bind to. This name should match the name you define in <code>web.xml</code> file.
<code>httpMethodRestrict</code>	<code>null</code>	Camel 2.11: Consumer only: Used to only allow consuming if the HttpMethod matches, such as GET/POST/PUT etc. From Camel 2.15 onwards multiple methods can be specified separated by comma.

Message Headers

Camel will apply the same Message Headers as the HTTP component.

Camel will also populate **all** `request.parameter` and `request.headers`. For example, if a client request has the URL, `http://myserver/myserver?orderid=123`, the exchange will contain a header named `orderid` with the value 123.

Usage

You can consume only from endpoints generated by the Servlet component. Therefore, it should be used only as input into your Camel routes. To issue HTTP requests against other HTTP endpoints, use the HTTP Component

Using Servlet 3.0 Async Mode

Available as of Camel 2.18

You can configure the servlet with an init-param to turn on async mode when using a Servlet 3.x container. There is a sample XML configuration below:

```
xml <servlet> <servlet-name>CamelServlet</servlet-name> <display-name>Camel Http Transport Servlet</display-name>
<servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class> <init-param> <param-
name>async</param-name> <param-value>true</param-value> </init-param> <load-on-startup>1</load-on-startup> <async-
supported>true</async-supported> </servlet>
```

Putting Camel JARs in the app server boot classpath

If you put the Camel JARs such as `camel-core`, `camel-servlet`, etc. in the boot classpath of your application server (eg usually in its lib directory), then mind that the servlet mapping list is now shared between multiple deployed Camel application in the app server.

Mind that putting Camel JARs in the boot classpath of the application server is generally not best practice!

So in those situations you **must** define a custom and unique servlet name in each of your Camel application, eg in the `web.xml` define:

```
xml <servlet> <servlet-name>MyServlet</servlet-name> <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class> <load-on-startup>1</load-on-startup>
</servlet> <servlet-mapping> <servlet-name>MyServlet</servlet-name> <url-pattern>/*</url-pattern> </servlet-mapping>
```

And in your Camel endpoints then include the servlet name as well

```
xml <route> <from uri="servlet:/foo?servletName=MyServlet"/> ... </route>
```

From **Camel 2.11** onwards Camel will detect this duplicate and fail to start the application. You can control to ignore this duplicate by setting the servlet init-parameter `ignoreDuplicateServletName` to true as follows:

```
xml <servlet> <servlet-name>CamelServlet</servlet-name> <display-name>Camel Http Transport Servlet</display-name>
<servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class> <init-param> <param-
name>ignoreDuplicateServletName</param-name> <param-value>true</param-value> </init-param> </servlet>
```

But its **strongly advised** to use unique servlet-name for each Camel application to avoid this duplication clash, as well any unforeseen side-effects.

Sample

From Camel 2.7 onwards it's easier to use Servlet in Spring web applications. See [Servlet Tomcat Example](#) for details.

In this sample, we define a route that exposes a HTTP service at `http://localhost:8080/camel/services/hello`.

First, you need to publish the `CamelHttpTransportServlet` through the normal Web Container, or OSGi Service.

Use the `web.xml` file to publish the `CamelHttpTransportServlet` as follows:

```
{snippet:id=web|lang=xml|url=camel/trunk/components/camel-
servlet/src/test/resources/org/apache/camel/component/servlet/web.xml}Then you can define your route as follows:
{snippet:id=route|lang=java|url=camel/trunk/components/camel-
servlet/src/test/java/org/apache/camel/component/servlet/HttpClientRouteTest.java}
```

Specify the relative path for `camel-servlet` endpoint

Since we are binding the `Http transport` with a published servlet, and we don't know the servlet's application context path, the `camel-servlet` endpoint uses the relative path to specify the endpoint's URL. A client can access the `camel-servlet` endpoint through the servlet publish address: (`"http://localhost:8080/camel/services"`) + `RELATIVE_PATH` (`"/hello"`).

Sample when using Spring 3.x

See Servlet Tomcat Example

Sample when using Spring 2.x

When using the Servlet component in a Camel/Spring application it's often required to load the Spring ApplicationContext after the Servlet component has started. This can be accomplished by using Spring's ContextLoaderServlet instead of ContextLoaderListener. In that case you'll need to start ContextLoaderServlet after CamelHttpTransportServlet like this:

```
xml <web-app> <servlet> <servlet-name>CamelServlet</servlet-name> <servlet-class>
org.apache.camel.component.servlet.CamelHttpTransportServlet </servlet-class> <load-on-startup>1</load-on-startup>
</servlet> <servlet> <servlet-name>SpringApplicationContext</servlet-name> <servlet-class>
org.springframework.web.context.ContextLoaderServlet </servlet-class> <load-on-startup>2</load-on-startup> </servlet> <web-
app>
```

Sample when using OSGi

From **Camel 2.6.0**, you can publish the CamelHttpTransportServlet as an OSGi service with help of SpringDM like this.

```
{snippet:id=service|lang=xml|url=camel/trunk/tests/camel-itest-
osgi/src/test/resources/org/apache/camel/itest/osgi/servlet/ServletServiceContext.xml}Then use this service in your camel route
like this:{snippet:id=camelContext|lang=xml|url=camel/trunk/tests/camel-itest-
osgi/src/test/resources/org/apache/camel/itest/osgi/servlet/CamelServletWithServletServiceContext.xml}For versions prior to
Camel 2.6 you can use an Activator to publish the CamelHttpTransportServlet on the OSGi
platform{snippet:id=activator|lang=java|url=camel/trunk/tests/camel-itest-
osgi/src/test/java/org/apache/camel/itest/osgi/servlet/support/ServletActivator.java}Endpoint See Also
```

- Servlet Tomcat Example
- Servlet Tomcat No Spring Example
- HTTP
- Jetty

SHIRO SECURITY COMPONENT

Available as of Camel 2.5

The **shiro-security** component in Camel is a security focused component, based on the Apache Shiro security project.

Apache Shiro is a powerful and flexible open-source security framework that cleanly handles authentication, authorization, enterprise session management and cryptography. The objective of the Apache Shiro project is to provide the most robust and comprehensive application security framework available while also being very easy to understand and extremely simple to use.

This camel shiro-security component allows authentication and authorization support to be applied to different segments of a camel route.

Shiro security is applied on a route using a Camel Policy. A Policy in Camel utilizes a strategy pattern for applying interceptors on Camel Processors. It offering the ability to apply cross-cutting concerns (for example. security, transactions etc) on sections/segments of a camel route.

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

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

Shiro Security Basics

To employ Shiro security on a camel route, a ShiroSecurityPolicy object must be instantiated with security configuration details (including users, passwords, roles etc). This object must then be applied to a camel route. This ShiroSecurityPolicy Object may also be registered in the Camel registry (JNDI or ApplicationContextRegistry) and then utilized on other routes in the Camel Context.

Configuration details are provided to the ShiroSecurityPolicy using an Ini file (properties file) or an Ini object. The Ini file is a standard Shiro configuration file containing user/role details as shown below

```
[users]
# user 'ringo'; ringo with password 'starr' and the 'sec-level1' role
ringo = starr, sec-level1
george = harrison, sec-level1
john = lennon, sec-level1
paul = mccartney, sec-level1

[roles]
# 'sec-level3' role has all permissions, indicated by the
# wildcard '*' and 'sec-level3'
sec-level3 = *
```

```
# The &#39;sec-level2&#39; role can do anything with access of permission
# readonly (*) to help
sec-level2 = zone1:*

# The &#39;sec-level1&#39; role can do anything with access of permission
# readonly
sec-level1 = zone1:readonly:*
```

Instantiating a ShiroSecurityPolicy Object

A ShiroSecurityPolicy object is instantiated as follows

```
private final String iniResourcePath = "classpath:shiro.ini";
private final byte[] passPhrase = {
    (byte) 0x08, (byte) 0x09, (byte) 0x0A, (byte) 0x0B,
    (byte) 0x0C, (byte) 0x0D, (byte) 0x0E, (byte) 0x0F,
    (byte) 0x10, (byte) 0x11, (byte) 0x12, (byte) 0x13,
    (byte) 0x14, (byte) 0x15, (byte) 0x16, (byte) 0x17};
List<Permission> permissionsList = new ArrayList<Permission>();
Permission permission = new WildcardPermission("zone1:readwrite:*");
permissionsList.add(permission);

final ShiroSecurityPolicy securityPolicy =
    new ShiroSecurityPolicy(iniResourcePath, passPhrase, true, permissionsList);
```

ShiroSecurityPolicy Options

Name	Default Value	Type	Description
iniResourcePath or ini	none	Resource String or Ini Object	A mandatory Resource String for the iniResourcePath or an instance of an Ini object must be passed to the security policy. Resources can be acquired from the iniResourcePath, or an instance of Ini, provided with the necessary key for the resources, for e.g., classpath:shiro.ini
passPhrase	An AES 128 byte[] 16bytes AES	byte[]	A passPhrase to decrypt ShiroSecurityToken(s) sent along with Message Exchanges
alwaysAuthenticate	true	boolean	Setting to ensure re-authentication on every individual request. If set to false, the user is authenticated and locked such that only requests from the same user going forward are authenticated.
permissionsList	none	List<Permission>	A List of permissions required in order for an authenticated user to be authorized to perform further action i.e continue further on the route. If no Permission list or Roles List (see below) is provided to the ShiroSecurityPolicy object, then authorization is deemed as not required. Note that the default is alwaysAuthenticate=true, so if no permission list is provided, then authorization is deemed as not required.
rolesList	none	List<String>	Camel 2.12: A List of roles required in order for an authenticated user to be authorized to perform further action i.e continues further on the route. If no roles list or permissions list (see above) is provided to the ShiroSecurityPolicy object, then authorization is deemed as not required. Note that the default is alwaysAuthenticate=true, so if no roles list is provided, then authorization is deemed as not required.
cipherService	AES	org.apache.shiro.crypto.CipherService	Shiro ships with AES & Blowfish based CipherServices. You may use one these or pass in your own Cipher implementation
base64	false	boolean	Camel 2.12: To use base64 encoding for the security token header, which allows transferring the header over JMS etc. This option must also be set on ShiroSecurityTokenEncoder as well.
allPermissionsRequired	false	boolean	Camel 2.13: The default is that authorization is granted if any of the Permission Objects in the permissionsList parameter are applicable. Set this to true to require all of the Permissions to be met.
allRolesRequired	false	boolean	Camel 2.13: The default is that authorization is granted if any of the roles in the rolesList parameter are applicable. Set this to true to require all of the roles to be met.

Applying Shiro Authentication on a Camel Route

The ShiroSecurityPolicy, tests and permits incoming message exchanges containing a encrypted SecurityToken in the Message Header to proceed further following proper authentication. The SecurityToken object contains a Username/Password details that are used to determine where the user is a valid user.

```
protected RouteBuilder createRouteBuilder() throws Exception {
    final ShiroSecurityPolicy securityPolicy =
        new ShiroSecurityPolicy("classpath:shiro.ini", passPhrase);

    return new RouteBuilder() {
        public void configure() {
            onException(UnknownAccountException.class).
                to("mock:authenticationException");
            onException(IncorrectCredentialsException.class).
                to("mock:authenticationException");
            onException(LockedAccountException.class).
                to("mock:authenticationException");
            onException(AuthenticationException.class).
                to("mock:authenticationException");

            from("direct:secureEndpoint").
                to("log:incoming payload").
                policy(securityPolicy).
                to("mock:success");
        }
    };
}
```

Applying Shiro Authorization on a Camel Route

Authorization can be applied on a camel route by associating a Permissions List with the ShiroSecurityPolicy. The Permissions List specifies the permissions necessary for the user to proceed with the execution of the route segment. If the user does not have the proper permission set, the request is not authorized to continue any further.

```
protected RouteBuilder createRouteBuilder() throws Exception {
    final ShiroSecurityPolicy securityPolicy =
        new ShiroSecurityPolicy("./src/test/resources/securityconfig.ini", passPhrase);

    return new RouteBuilder() {
        public void configure() {
            onException(UnknownAccountException.class).
```

```

        to("mock:authenticationException");
    onException(IncorrectCredentialsException.class)
        to("mock:authenticationException");
    onException(LockedAccountException.class).
        to("mock:authenticationException");
    onException(AuthenticationException.class).
        to("mock:authenticationException");

from("direct:secureEndpoint").
    to("log:incoming payload").
    policy(securityPolicy).
    to("mock:success");
}
}
}
}
```

Creating a ShiroSecurityToken and injecting it into a Message Exchange

A ShiroSecurityToken object may be created and injected into a Message Exchange using a Shiro Processor called ShiroSecurityTokenInjector. An example of injecting a ShiroSecurityToken using a ShiroSecurityTokenInjector in the client is shown below.

```
ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "starr");
ShiroSecurityTokenInjector shiroSecurityTokenInjector =
    new ShiroSecurityTokenInjector(shiroSecurityToken, passPhrase);

from("direct:client").
    process(shiroSecurityTokenInjector).
    to("direct:secureEndpoint");
```

Sending Messages to routes secured by a ShiroSecurityPolicy

Messages and Message Exchanges sent along the camel route where the security policy is applied need to be accompanied by a SecurityToken in the Exchange Header. The SecurityToken is an encrypted object that holds a Username and Password. The SecurityToken is encrypted using AES 128 bit security by default and can be changed to any cipher of your choice.

Given below is an example of how a request may be sent using a ProducerTemplate in Camel along with a SecurityToken.

```
    @Test
    public void testSuccessfulShiroAuthenticationWithNoAuthorization() throws Exception {
        //Incorrect password
        ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "stirr");

        // TestShiroSecurityTokenInjector extends ShiroSecurityTokenInjector
        TestShiroSecurityTokenInjector shiroSecurityTokenInjector =
            new TestShiroSecurityTokenInjector(shiroSecurityToken, passPhrase);

        successEndpoint.expectedMessageCount(1);
        failureEndpoint.expectedMessageCount(0);

        template.send("direct:secureEndpoint", shiroSecurityTokenInjector);

        successEndpoint.assertIsSatisfied();
        failureEndpoint.assertIsSatisfied();
    }
```

Sending Messages to routes secured by a ShiroSecurityPolicy (much easier from Camel 2.12 onwards)

From Camel 2.12 onwards its even easier as you can provide the subject in two different ways.

Using ShiroSecurityToken

You can send a message to a Camel route with a header of key `ShiroSecurityConstants.SHIRO_SECURITY_TOKEN` of the type `org.apache.camel.component.shiro.security.ShiroSecurityToken` that contains the username and password. For example:

```
ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "starr");
template.sendBodyAndHeader("direct:secureEndpoint", "Beatle Mania", ShiroSecurityConstants.SHIRO_SECURITY_TOKEN, shiroSecurityToken);
```

You can also provide the username and password in two different headers as shown below.

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put(ShiroSecurityConstants.SHIRO_SECURITY_USERNAME, "ringo");
headers.put(ShiroSecurityConstants.SHIRO_SECURITY_PASSWORD, "star");
template.sendBodyAndHeaders("direct:secureEndpoint", "Beatle Mania", headers)
```

When you use the username and password headers, then the ShiroSecurityPolicy in the Camel route will automatically transform those into a single header with key ShiroSecurityConstants.SHIRO_SECURITY_TOKEN with the token. Then token is either a ShiroSecurityToken instance, or a base64 representation as a String (the latter is when you have set base64=true).

SIP COMPONENT

Available as of Camel 2.5

The **sip** component in Camel is a communication component, based on the Jain SIP implementation (available under the JCP license).

Session Initiation Protocol (SIP) is an IETF-defined signaling protocol, widely used for controlling multimedia communication sessions such as voice and video calls over Internet Protocol (IP). The SIP protocol is an Application Layer protocol designed to be independent of the underlying transport layer; it can run on Transmission Control Protocol (TCP), User Datagram Protocol (UDP) or Stream Control Transmission Protocol (SCTP).

The Jain SIP implementation supports TCP and UDP only.

The Camel SIP component **only** supports the SIP Publish and Subscribe capability as described in the RFC3903 - Session Initiation Protocol (SIP) Extension for Event

This camel component supports both producer and consumer endpoints.

Camel SIP Producers (Event Publishers) and SIP Consumers (Event Subscribers) communicate event & state information to each other using an intermediary entity called a SIP Presence Agent (a stateful brokering entity).

For SIP based communication, a SIP Stack with a listener **must** be instantiated on both the SIP Producer and Consumer (using separate ports if using localhost). This is necessary in order to support the handshakes & acknowledgements exchanged between the SIP Stacks during communication.

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

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

URI format

The URI scheme for a sip endpoint is as follows:

```
sip://johndoe@localhost:9999[?options]
sips://johndoe@localhost:9999[?options]
```

This component supports producer and consumer endpoints for both TCP and UDP.

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

Options

The SIP Component offers an extensive set of configuration options & capability to create custom stateful headers needed to propagate state via the SIP protocol.

Name	Default Value	Description
stackname	NAME_NOT_SET	Name of the SIP Stack instance associated with an SIP Endpoint.
transport	tcp	Setting for choice of transport protocol. Valid choices are "tcp" or "udp".
fromUser		Username of the message originator. Mandatory setting unless a registry based custom FromHeader is specified.
fromHost		Hostname of the message originator. Mandatory setting unless a registry based FromHeader is specified
fromPort		Port of the message originator. Mandatory setting unless a registry based FromHeader is specified
toUser		Username of the message receiver. Mandatory setting unless a registry based custom ToHeader is specified
toHost		Hostname of the message receiver. Mandatory setting unless a registry based custom ToHeader is specified
toPort		Portname of the message receiver. Mandatory setting unless a registry based custom ToHeader is specified
maxForwards	0	the number of intermediaries that may forward the message to the message receiver. Optional setting. May alternatively be set using as registry based MaxForwardsHeader
eventId		Setting for a String based event id. Mandatory setting unless a registry based FromHeader is specified
eventHeaderName		Setting for a String based event id. Mandatory setting unless a registry based FromHeader is specified
maxMessageSize	1048576	Setting for maximum allowed Message size in bytes
cacheConnections	false	Should connections be cached by the SipStack to reduce cost of connection creation. This is useful if the connection is used for long running conversations.
consume	false	This setting is used to determine whether the kind of header (FromHeader, ToHeader etc) that needs to be created for this endpoint
contentSubType	xml	Setting for contentSubType can be set to any valid MimeSubType
receiveTimeoutMillis	10000	Setting for specifying amount of time to wait for a Response and/or Acknowledgement can be received from another SIP stack
useSoterForAllUris	false	This setting is used when requests are sent to the Presence Agent via a proxy.
msgExpiration	3600	The amount of time a message received at an endpoint is considered valid
presenceAgent	false	This setting is used to distinguish between a Presence Agent & a consumer. This is due to the fact that the SIP Camel component ships with a basic Presence Agent (for testing purposes only).

Registry based Options

SIP requires a number of headers to be sent/received as part of a request. These SIP header can be enlisted in the Registry, such as in the Spring XML file.

The values that could be passed in, are the following:

Name	Description
fromHeader	a custom Header object containing message originator settings. Must implement the type javax.sip.header.FromHeader
toHeader	a custom Header object containing message receiver settings. Must implement the type javax.sip.header.ToHeader
viaHeaders	List of custom Header objects of the type javax.sip.header.ViaHeader. Each ViaHeader containing a proxy address for request forwarding. (Note this header is automatically updated by each proxy when the request arrives at its listener)

contentTypeHeader	a custom Header object containing message content details. Must implement the type javax.sip.header.ContentTypeHeader
callIdHeader	a custom Header object containing call id details. Must implement the type javax.sip.header.CallIdHeader
maxForwardsHeader	a custom Header object containing details on maximum proxy forwards. This header places a limit on the viaHeaders possible. Must implement the type javax.sip.header.MaxForwardsHeader
eventHeader	a custom Header object containing event details. Must implement the type javax.sip.header.EventHeader
contactHeader	an optional custom Header object containing verbose contact details (email, phone number etc). Must implement the type javax.sip.header.ContactHeader
expiryHeader	a custom Header object containing message expiration details. Must implement the type javax.sip.header.ExpiresHeader
extensionHeader	a custom Header object containing user/application specific details. Must implement the type javax.sip.header.ExtensionHeader

Sending Messages to/from a SIP endpoint

Creating a Camel SIP Publisher

In the example below, a SIP Publisher is created to send SIP Event publications to a user "agent@localhost:5152". This is the address of the SIP Presence Agent which acts as a broker between the SIP Publisher and Subscriber

- using a SIP Stack named client
- using a registry based eventHeader called evtHdrName
- using a registry based eventId called evtid
- from a SIP Stack with Listener set up as user2@localhost:3534
- The Event being published is EVENT_A
- A Mandatory Header called REQUEST_METHOD is set to Request.PUBLISH thereby setting up the endpoint as a Event publisher"

```
producerTemplate.sendBodyAndHeader(
    "sip://agent@localhost:5152?stackName=client&eventHeaderName=evtHdrName&eventId=evtid&fromUser=user2&fromHost=localhost&fromPort=3534",
    "EVENT_A",
    "REQUEST_METHOD",
    Request.PUBLISH);
```

Creating a Camel SIP Subscriber

In the example below, a SIP Subscriber is created to receive SIP Event publications sent to a user "johndoe@localhost:5154"

- using a SIP Stack named Subscriber
- registering with a Presence Agent user called agent@localhost:5152
- using a registry based eventHeader called evtHdrName. The evtHdrName contains the Event which is set to "Event_A"
- using a registry based eventId called evtid

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            // Create PresenceAgent
            from("sip://agent@localhost:5152?stackName=PresenceAgent&presetenceAgent=true&eventHeaderName=evtHdrName&eventId=evtid")
                .to("mock:neverland");

            // Create Sip Consumer(Event Subscriber)
            from("sip://johndoe@localhost:5154?stackName=Subscriber&toUser=agent&toHost=localhost&toPort=5152&eventHeaderName=evtHdrName&event"
                .to("log:ReceivedEvent?level=DEBUG")
                .to("mock:notification");
        }
    };
}
```

The Camel SIP component also ships with a Presence Agent that is meant to be used for Testing and Demo purposes only. An example of instantiating a Presence Agent is given above.

Note that the Presence Agent is set up as a user agent@localhost:5152 and is capable of communicating with both Publisher as well as Subscriber. It has a separate SIP stackName distinct from Publisher as well as Subscriber. While it is set up as a Camel Consumer, it does not actually send any messages along the route to the endpoint "mock:neverland".

SMPP COMPONENT

CamelSmppFinalStatusAvailable as of Camel 2.2

This component provides access to an SMSC (Short Message Service Center) over the SMPP protocol to send and receive SMS. The JSmpp library is used for the protocol implementation.

The Camel component currently operates as an ESME (External Short Messaging Entity) and not as an SMSC itself.

The SMS Router project provides an excellent example of an SMS Router daemon based on the Camel framework, routing messages between JMS queues (ActiveMQ) and the SMPP network. Run multiple instances in parallel for high-availability.

Starting with **Camel 2.9** you are also able to execute ReplaceSm, QuerySm, SubmitMulti, CancelSm and DataSm.

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

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

SMS limitations

SMS is neither reliable or secure. Users who require reliable and secure delivery may want to consider using the XMPP or SIP components instead, combined with a smartphone app supporting the chosen protocol.

- Reliability: although the SMPP standard offers a range of feedback mechanisms to indicate errors, non-delivery and confirmation of delivery it is not uncommon for mobile networks to hide or simulate these responses. For example, some networks automatically send a delivery confirmation for every message even if the destination number is invalid or not switched on. Some networks silently drop messages if they think they are spam. Spam detection rules in the network may be very crude, sometimes more than 100 messages per day from a single sender may be considered spam.
- Security: there is basic encryption for the last hop from the radio tower down to the recipient handset. SMS messages are not encrypted or authenticated in any other part of the network. Some operators allow staff in retail outlets or call centres to browse through the SMS message histories of their customers. Message sender identity can be easily forged. Regulators and even the mobile telephone industry itself has cautioned against the use of SMS in two-factor authentication schemes and other purposes where security is important.

While the Camel component makes it as easy as possible to send messages to the SMS network, it can not offer an easy solution to these problems.

DATA CODING, ALPHABET AND INTERNATIONAL CHARACTER SETS

Data coding and alphabet can be specified on a per-message basis. Default values can be specified for the endpoint. It is important to understand the relationship between these options and the way the component acts when more than one value is set.

Data coding is an 8 bit field in the SMPP wire format.

Alphabet corresponds to bits 0-3 of the data coding field. For some types of message, where a message class is used (by setting bit 5 of the data coding field), the lower two bits of the data coding field are not interpreted as alphabet and only bits 2 and 3 impact the alphabet.

Furthermore, current version of the JSMPP library only seems to support bits 2 and 3, assuming that bits 0 and 1 are used for message class. This is why the Alphabet class in JSMPP doesn't support the value 3 (binary 0011) which indicates ISO-8859-1.

Although JSMPP provides a representation of the message class parameter, the Camel component doesn't currently provide a way to set it other than manually setting the corresponding bits in the data coding field.

When setting the data coding field in the outgoing message, the Camel component considers the following values and uses the first one it can find:

- the data coding specified in a header
- the alphabet specified in a header
- the data coding specified in the endpoint configuration (URI parameter)

Older versions of Camel had bugs in support for international character sets. This feature only worked when a single encoding was used for all messages and was troublesome when users wanted to change it on a per-message basis. Users who require this to work should ensure their version of Camel includes the fix for

JIRA Issues Macro: Unable to locate JIRA server for this macro. It may be due to Application Link configuration.

In addition to trying to send the data coding value to the SMSC, the Camel component also tries to analyze the message body, convert it to a Java String (Unicode) and convert that to a byte array in the corresponding alphabet. When deciding which alphabet to use in the byte array, the Camel SMPP component does not consider the data coding value (header or configuration), it only considers the specified alphabet (from either the header or endpoint parameter).

If some characters in the String can't be represented in the chosen alphabet, they may be replaced by the question mark (?) symbol. Users of the API may want to consider checking if their message body can be converted to ISO-8859-1 before passing it to the component and if not, setting the alphabet header to request UCS-2 encoding. If the alphabet and data coding options are not specified at all then the component may try to detect the required encoding and set the data coding for you.

The list of alphabet codes are specified in the SMPP specification v3.4, section 5.2.19. One notable limitation of the SMPP specification is that there is no alphabet code for explicitly requesting use of the GSM 3.38 (7 bit) character set. Choosing the value 0 for the alphabet selects the SMSC *default* alphabet, this usually means GSM 3.38 but it is not guaranteed. The SMPP gateway Nexmo actually allows the default to be mapped to any other character set with a control panel option. It is suggested that users check with their SMSC operator to confirm exactly which character set is being used as the default.

Message splitting and throttling

After transforming a message body from a String to a byte array, the Camel component is also responsible for splitting the message into parts (within the 140 byte SMS size limit) before passing it to JSMPP. This is completed automatically.

If the GSM 3.38 alphabet is used, the component will pack up to 160 characters into the 140 byte message body. If an 8 bit character set is used (e.g. ISO-8859-1 for western Europe) then 140 characters will be allowed within the 140 byte message body. If 16 bit UCS-2 encoding is used then just 70 characters fit into each 140 byte message.

Some SMSC providers implement throttling rules. Each part of a message that has been split may be counted separately by the provider's throttling mechanism. The Camel Throttler component can be useful for throttling messages in the SMPP route before handing them to the SMSC.

URI format

```
smpp://[username@]hostname[:port][?options]
smpps://[username@]hostname[:port][?options]
```

If no **username** is provided, then Camel will provide the default value `smppclient`.

If no **port** number is provided, then Camel will provide the default value `2775`.

Camel 2.3: If the protocol name is "smpps", camel-smpp with try to use SSLSocket to init a connection to the server.

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

URI Options

Name	Default Value	Description
<code>password</code>	<code>password</code>	Specifies the password to use to log in to the SMSC.
<code>systemtype</code>	<code>CPT</code>	This parameter is used to categorize the type of ESME (External Short Message Entity) that is binding to the SMSC (max. 13 characters). Camel 2.11 Defines the data coding according the SMPP 3.4 specification, section 5.2.19. (Prior to Camel 2.9, this option is also supported.) Example data encodings are: o: SMSC Default Alphabet 1: Latin 1 (ISO-8859-1) 2: Latin 2 (ISO-8859-2) 3: Latin 3 (ISO-8859-3) 4: Latin 4 (ISO-8859-4) 5: Latin 5 (ISO-8859-5) 6: Latin 6 (ISO-8859-6) 7: Latin 7 (ISO-8859-7) 8: Latin 8 (ISO-8859-8 binary) 9: UCS2 (ISO/IEC-10646) 13: Extended Kanji JISX 0212-1990
<code>datacoding</code>	<code>0</code>	Camel 2.5 Defines encoding of data according the SMPP 3.4 specification, section 5.2.19. This option is mapped to <code>Alphabet.java</code> and used to create the <code>byte[]</code> which is send to the SMSC. Example data encodings are: o: SMSC Default Alphabet 1: 8 bit Alphabet 2: UCS2 Alphabet
<code>alphabet</code>	<code>0</code>	Camel 2.5 Defines encoding of data according the SMPP 3.4 specification, section 5.2.19. This option is mapped to <code>Alphabet.java</code> and used to create the <code>byte[]</code> which is send to the SMSC.
<code>encoding</code>	<code>ISO-8859-1</code>	only for SubmitSm, ReplaceSm and SubmitMulti Defines the encoding scheme of the short message user data.
<code>enquireLinkTime</code>	<code>5000</code>	Defines the interval in milliseconds between the confidence checks. The confidence check is used to test the communication path between an ESME and an SMSC.
<code>transactionTimer</code>	<code>10000</code>	Defines the maximum period of inactivity allowed after a transaction, after which an SMPP entity may assume that the session is no longer active. This timer may be active on either communicating SMPP entity (i.e. SMSC or ESME).
<code>initialReconnectDelay</code>	<code>5000</code>	Defines the initial delay in milliseconds after the consumer/producer tries to reconnect to the SMSC, after the connection was lost.
<code>reconnectDelay</code>	<code>5000</code>	Defines the interval in milliseconds between the reconnect attempts, if the connection to the SMSC was lost and the previous was not successful.
<code>registeredDelivery</code>	<code>1</code>	only for SubmitSm, ReplaceSm, SubmitMulti and DataSm Is used to request an SMSC delivery receipt and/or SME originated acknowledgements. The following values are defined: 0: No delivery receipt requested 1: SMSC delivery receipt requested where final delivery outcome is success or failure 2: SMSC delivery receipt requested where the final delivery outcome is delivery failure
<code>serviceType</code>	<code>CMT</code>	The service type parameter can be used to indicate the SMS Application service associated with the message. The following generic service_types are defined: cmt: Cellular Messaging cmr: Cellular Management vms: Voice Mail Notification vma: Voice Mail Alerting wms: Wireless Message Service Protocol ussd: Unstructured Supplementary Services Data
<code>sourceaddr</code>	<code>1616</code>	Defines the address of SME (Short Message Entity) which originated this message.
<code>destaddr</code>	<code>1712</code>	only for SubmitSm, SubstituteSm, CancelSm and DataSm Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
<code>sourceTon</code>		The type of number (TON) to be used in the SME originator address parameters. The following TON values are defined: 0: Unknown 1: International 2: National 3: Network Specific 4: Subscriber Number 5: Broadcast 6: Abbreviated
<code>destAddrton</code>	<code>0</code>	only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the type of number (TON) to be used in the SME destination address parameters. Same as the <code>sourceTon</code> values defined above.
<code>sourceNpi</code>		Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. The following NPI values are defined: 0: Unknown 1: ISDN (E163/E164) 2: Data (X.121) 3: Telex (F.69) 4: Mobile (E.212) 8: National 9: Private 10: INMARSAT 13: Internet (IP) 18: WAP Client Id (to be defined by WAP Forum)
<code>destAddrnpi</code>	<code>0</code>	only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. Same as the <code>sourceNpi</code> values defined above.
<code>priorityFlag</code>	<code>1</code>	only for SubmitSm and SubmitMulti Allows the originating SME to assign a priority level to the short message. Four Priority Levels are supported: 0: Level 1 priority 1: Level 2 priority 2: Level 3 priority 3: Level 4 priority
<code>replaceIfPresentFlag</code>	<code>0</code>	only for SubmitSm and SubmitMulti Used to request the SMSC to replace a previously submitted message, that is still pending delivery. The SMSC will replace an existing message provided that the source address, destination address and service type match the same fields in the new message. The following replaceIfPresent flag values are defined: 0: Don't replace 1: Replace
<code>typeIndicator</code>	<code>1</code>	Defines the type of number (TON) to be used in the SME originator address parameters.
<code>numberingPlanIndicator</code>	<code>0</code>	Defines the numeric plan indicator (NPI) to be used in the SME. Use the <code>sourceTon</code> values defined above.
<code>lazySessionCreation</code>	<code>false</code>	Camel 2.8 onwards Sessions can be lazily created to avoid exceptions, if the SMSC is not available when the Camel producer is started.
<code>httpProxyHost</code>	<code>null</code>	Camel 2.9.1: If your HTTP proxy requires basic authentication, set the attribute to the username required for your HTTP proxy.
<code>httpProxyPort</code>	<code>3128</code>	Camel 2.9.1: If your HTTP proxy requires basic authentication, set the attribute to the port required for your HTTP proxy.
<code>httpProxyPassword</code>	<code>null</code>	Camel 2.9.1: If your HTTP proxy requires basic authentication, set the attribute to the password required for your HTTP proxy.
<code>sessionStateListener</code>	<code>null</code>	Camel 2.9.1: You can refer to a org.apache.camel.component.smpp.SessionStateListener in the Registry to receive callbacks when the session state changed.
<code>addressRange</code>	<code>=</code>	Camel 2.11: You can specify the address range for the SmppConsumer as defined in section 5.2.7 of the SMPP 3.4 specification. The SmppConsumer will receive messages only from SMSC's which target an address (MSIDN or IP address) within this range.
<code>splittingPolicy</code>	<code>ALLOW</code>	Camel 2.14.1 and 2.15.0: You can specify a policy for handling long messages: • ALLOW - the default, long messages are split to 140 bytes per message

- TRUNCATE - long messages are split and only the first fragment will be sent to the SMSC. Some carriers drop subsequent fragments so this reduces load on the SMPP connection sending parts of a message that will never be delivered.
- REJECT - if a message would need to be split, it is rejected with an SMPP NegativeResponseException and the reason code signifying the message is too long.

proxyHeaders	map	Camel 2.17: These headers are used to define the destination address(es) and type of number (TON) to be used in the SME destination address parameters. Use the sourceaddrFrom URI option
maxReconnect	2147483647	Camel 2.19: Defines the maximum number of attempts to reconnect to the SMSC, if SMSC returns a negative bind response

You can have as many of these options as you like.

```
smpc://smppclient@localhost:2775?password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=consumer
```

Producer Message Headers

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

Header	Type	Description
CamelSmpBestAddr	List<String>	only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the destination SME address(es). For mobile terminated messages, this is the directory number of the recipient MS. Is must be a List<String> for SubmitMulti, and a String otherwise.
CamelSmpPstesAddTrn	Byte	only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the type of number (TON) to be used in the SME destination address parameters. Use the sourceaddrFrom URI option values defined above.
CamelSmpPstesAddRp1	Byte	only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. Use the sourceaddrFrom URI option sourceaddrFrom values defined above.
CamelSmpSourceAddr	String	Defines the address of SME (Short Message Entity) which originated this message.
CamelSmpSourceAddTrn	Byte	Defines the type of number (TON) to be used in the SME originator address parameters. Use the sourceaddrFrom URI option values defined above.
CamelSmpSourceAddRp1	Byte	Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. Use the sourceaddrFrom URI option sourceaddrFrom values defined above.
CamelSmpPriority	String	Only for ReplaceSm, ReplaceMulti and DataSm. Is used to request an SMSC delivery receipt and/or originate acknowledgements. Use the URI option CamelSmpPriorityLevel above.
CamelSmpReplaceDelivery	Byte	Only for ReplaceSm, ReplaceMulti and DataSm. Is used to request an SMSC delivery receipt and/or originate acknowledgements. Use the URI option CamelSmpPriorityLevel above.
CamelSmpPriorityFlag	Byte	only for SubmitSm and SubmitMulti Allows the originating SME to assign a priority level to the short message. Use the URI option priorityFlag settings above.
CamelSmpScheduleDeliveryTime	Date	only for SubmitSm, SubmitMulti and ReplaceSm. The scheduled time at which the message delivery should be first attempted. It defines either the absolute date and time or relative time from the current SMSC time at which delivery of this message will be attempted by the SMSC. It can be specified in either absolute time format or relative time format. The encoding of a time format is specified in chapter 7.1.1, in the smpp specification v3.4.
CamelSmpValidityPeriod	String/Date	only for SubmitSm, SubmitMulti and ReplaceSm. The validity period parameter indicates the SMSC expiration time, after which the message should be discarded if not delivered to the destination. If provided as date, it's interpreted as absolute time. Camel 2.9 onwards: It can be defined in absolute time format or relative time format if you provide it as string as specified in chapter 7.1.1, in the smpp specification v3.4.
CamelSmpReplaceIfPresentFlag	Byte	Only for SubmitSm and SubmitMulti. The replaceIfPresent flag parameter is used to request the SMSC to replace a previously submitted message, that is still pending delivery. The SMSC will replace an existing message provided that the source address, destination address and service type match the same fields in the new message. The following values are defined: 0: Do not replace 1: Replace
CamelSmpAphabet / CamelSmpDataCoding	Byte	Camel 2.5 For SubmitSm, SubmitMulti and ReplaceSm (Prior to Camel 2.9 use CamelSmpCoding instead of CamelSmpAphabet.) The data coding according to the SMPP 3.4 specification, section 5.2.19. Use the URI option alphabet settings above.
CamelSmpOptionalParameters	Map<String, String>	Deprecated and will be removed in Camel 2.13.0.0. Camel 2.10.6 and 2.11.1 onwards and only for SubmitSm, SubmitMulti and DataSm The optional parameters send back by the SMSC.
CamelSmpOptionalParameter	Map<Short, Object>	Camel 2.10.7 and 2.11.2 onwards and only for SubmitSm, SubmitMulti and DataSm The optional parameter which are send to the SMSC. The value is converted in the following way: String > org.jsmpp.bean.optionalParameter.String byte[] > org.jsmpp.bean.optionalParameter.octetString Byte > org.jsmpp.bean.optionalParameter.byte int > org.jsmpp.bean.optionalParameter.int Short > org.jsmpp.bean.optionalParameter.short null > org.jsmpp.bean.optionalParameter.null
CamelSmpEncoding	String	Camel 2.14.1 and Camel 2.15.0 onwards and only for SubmitSm, SubmitMulti and DataSm. Specifies the encoding (character set name) of the bytes in the message body. If the message body is a string then this is not relevant because Java Strings are always Unicode. If the body is a byte array then this header can be used to indicate that it is ISO-8859-1 or another character encoding. This header is only relevant for the message body, not the message header.
CamelSmpSplittingPolicy	String	Camel 2.14.1 and Camel 2.15.0 onwards and only for SubmitSm, SubmitMulti and DataSm. Specifies the policy for message splitting for this exchange. Possible values are described in the endpoint configuration parameter splittingPolicy.

The following message headers are used by the SMPP producer to set the response from the SMSC in the message header

Header	Type	Description
CamelSmpId	List<String>/String	The id to identify a submitted short message(s) for later use. From Camel 2.9.0: In case of a ReplaceSm, QuerySm, CancelSm and DataSm this header vault is a String. In case of a DataSm this header vault is a List<String>.
CamelSmpSentMessageCount	Integer	From Camel 2.9 onwards only for SubmitSm and SubmitMultiSm The total number of messages which has been sent.
CamelSmpError	Map<String, String>	From Camel 2.9 onwards only for SubmitSm/Smssm The errors which occurred by sending the short message(s) the form Map<String, List<Map<String, Object>>> (messageId : destAddr, address, error, errorCode).
CamelSmpOptionalParameters	Map<String, String>	Deprecated and will be removed only for DataSm. The optional parameters which are returned from the SMSC by sending the message.
CamelSmpOptionalParameter	Map<Short, Object>	From Camel 2.10.7, 2.11.2 onwards only for DataSm The optional parameter which are returned from the SMSC by sending the message. The key is the short code for the parameter. The value is converted in the following way: String > org.jsmpp.bean.optionalParameter.String byte[] > org.jsmpp.bean.optionalParameter.octetString Byte > org.jsmpp.bean.optionalParameter.byteString int > org.jsmpp.bean.optionalParameter.int Short > org.jsmpp.bean.optionalParameter.short null > org.jsmpp.bean.optionalParameter.null

Consumer Message Headers

The following message headers are used by the SMPP consumer to set the request data from the SMSC in the message header

Header	Type	Description
CamelSmpSequenceNumber	Integer	only for AlertNotification, DeliverSm and DataSm A sequence number allows a response PDU to be correlated with a request PDU. The associated SMPP response PDU must contain the same sequence number.
CamelSmpCommandId	Integer	only for AlertNotification, DeliverSm and DataSm The command id field identifies the particular SMPP PDU. For the complete list of defined values see chapter 5.12.1 in the specification.
CamelSmpSourceAddr	String	only for AlertNotification, DeliverSm and DataSm Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
CamelSmpSourceAddRp1	Byte	only for AlertNotification, DeliverSm and DataSm Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. Use the sourceaddrFrom URI option values defined above.
CamelSmpSourceAddTrn	Byte	only for AlertNotification Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
CamelSmpSourceAddRp1	Byte	only for AlertNotification Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. Use the sourceaddrFrom URI option values defined above.
CamelSmpSourceAddTrn	Byte	only for AlertNotification Defines the type of number (TON) to be used in the SME destination address parameters. Use the sourceaddrFrom URI option values defined above.
CamelSmpSeqId	String	only for smsc.DeliveryReceipt and DataSm The message ID allocated to the message by the SMSC when originally submitted.
CamelSmpDelivered	Integer	only for smsc.DeliveryReceipt Number of short messages delivered. This is only relevant where the original message was submitted to a distribution list. The value is padded with leading zeros if necessary.
CamelSmpPdate	Date	only for smsc.DeliveryReceipt The time and date at which the short message reached it's final state. The format is as follows: YYMMDDhhmm
CamelSmpStatus	DeliveryReceiptState	only for DeliveryReceipt: The final status of the message. The following values are defined: DELIVRD: Message is delivered to destination. EXPIRED: Message validity period has expired. DRAFT: Message is in draft state. UNDELIV: Message is in undeliverable state. ACCEPTD: Message is in accepted state. UNKNOWN: Message is in unknown state. REJECTD: Message is in rejected state.
CamelSmpCommandStatus	Integer	only for DataSm The Command status of the message.
CamelSmpError	String	only for smsc.DeliveryReceipt Where appropriate this may hold a Network specific error code or an SMSC error code for the attempted delivery of the message. These errors are Network or SMSC specific and are not included here.
CamelSmpSubmitDate	Date	only for smsc.DeliveryReceipt The time and date at which the short message was submitted. In the case of a message which has been replaced, this is the date that the

<code>camelSmppSubmitted</code>	<code>Integer</code>	original message was replaced. The format is as follows: YYMMDDhhmmss only for <code>smpp.DeliveryReceipt</code> . Number of messages originally submitted. This is only relevant when the original message was submitted to a distribution list. The value is padded with leading zeros if necessary.
<code>CamelSmppDestAddr</code>	<code>String</code>	only for <code>DeliverSm</code> and <code>DataSm</code> . Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
<code>camelSmppScheduleDeliveryTime</code>	<code>String</code>	only for <code>DeliverSm</code> and <code>DataSm</code> . Specifies the scheduled time at which the message delivery should be attempted. It defines either an absolute date and time or relative time from the current SMSC time at which delivery of this message will be attempted by the SMSC. It can be specified in either absolute time format or relative time format. The encoding of a time format is specified in Section 7.1.1 in the smpp specification v3.4.
<code>CamelSmppValidityPeriod</code>	<code>String</code>	only for <code>DeliverSm</code> The validity period parameter indicates the SMSC expiration time, after which the message should be discarded if not delivered to the destination. It can be defined in absolute time format or relative time format. The encoding of absolute and relative time format is specified in Section 7.1.1 in the smpp specification v3.4.
<code>CamelSmppServiceType</code>	<code>String</code>	only for <code>DeliverSm</code> and <code>DataSm</code> . The service type parameter indicates the SMS application service associated with the message.
<code>CamelSmppRegisteredDelivery</code>	<code>Byte</code>	only for <code>DeliverSm</code> and <code>DataSm</code> . The registered delivery parameter indicates the SMSC originated acknowledgements. Same values as in Producer header list above.
<code>CamelSmppDestAddrPn</code>	<code>Byte</code>	only for <code>DataSm</code> Defines the numeric plan indicator (NPn) in the destination address parameters. Use the <code>URI</code> option <code>sourceaddrpn!</code> values defined above.
<code>CamelSmppDestAddrTon</code>	<code>Byte</code>	only for <code>DataSm</code> Defines the type of number (TON) in the destination address parameters. Use the <code>sourceaddrton</code> <code>URI</code> option values defined above.
<code>CamelSmppMessageType</code>	<code>String</code>	Camel 2.6 onwards: Identifies the type of an incoming message. Alert notification: an SMSC alert notification dataSm: an SMSC data short message deliverSm: an SMSC delivered short message deliverSm: an SMSC delivered short message
<code>CamelSmppOptionalParameters</code>	<code>Map<String, Object></code>	Deprecated and will be removed in Camel 2.10.0.0.0 Camel 2.10.5 onwards and only for <code>DeliverSm</code> : The optional parameters send back by the SMSC.
<code>CamelSmppOptionalParameter</code>	<code>Map<Short, Object></code>	Camel 2.10.7, 2.11.2 onwards and only for <code>DeliverSm</code> : The optional parameters send back by the SMSC. The key is the short code for the optional parameter. The value is org.jsmpp.bean.optionalParameter.OctetString->String org.jsmpp.bean.optionalParameter.OctetString->byte[] org.jsmpp.bean.optionalParameter.Short->String org.jsmpp.bean.optionalParameter.Int->Integer org.jsmpp.bean.optionalParameter.Short->Short org.jsmpp.bean.optionalParameter.Null->Null

JSMPPL library

See the documentation of the JSMPPL Library for more details about the underlying library.

Exception handling

This component supports the general Camel exception handling capabilities

When an error occurs sending a message with `SubmitSm` (the default action), the `org.apache.camel.component.smpp.SmppException` is thrown with a nested exception, `org.jsmpp.extra.NegativeResponseException`. Call `NegativeResponseException.getCommandStatus()` to obtain the exact SMPP negative response code, the values are explained in the SMPP specification 3.4, section 5.1.3.

Camel 2.8 onwards: When the SMPP consumer receives a `deliverSm` or `dataSm` short message and the processing of these messages fails, you can also throw a `ProcessRequestException` instead of handle the failure. In this case, this exception is forwarded to the underlying JSMPPL library which will return the included error code to the SMSC. This feature is useful to e.g. instruct the SMSC to resend the short message at a later time. This could be done with the following lines of code:

```
from("smpp://smppclient@localhost:2775?password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=consumer")
.doTry()
.to("bean:dao?method=updateSmsState")
.doCatch(Exception.class)
.throwException(new ProcessRequestException("update of sms state failed", 100))
.end();
```

Please refer to the SMPP specification for the complete list of error codes and their meanings.

Samples

See the SMS Router source code for a full sample application. Consider integrating your application with SMS Router through queues rather than adding the SMPP code directly to your routes, building a more loosely-coupled architecture.

A route which sends an SMS using the Java DSL:

```
from("direct:start")
.to("smpp://smppclient@localhost:2775?
password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=producer");
```

A route which sends an SMS using the Spring XML DSL:

```
<route>
<from uri="direct:start"/>
<to uri="smpp://smppclient@localhost:2775?
password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=producer"/>
</route>
```

A route which receives an SMS using the Java DSL:

```
from("smpp://smppclient@localhost:2775?password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=consumer")
.to("bean:foo");
```

A route which receives an SMS using the Spring XML DSL:

```
<route>
<from uri="smpp://smppclient@localhost:2775?
password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=consumer"/>
<to uri="bean:foo"/>
</route>
```

SMSC simulator

If you need an SMSC simulator for your test, you can use the simulator provided by Logica.

Debug logging

This component has log level **DEBUG**, which can be helpful in debugging problems. If you use log4j, you can add the following line to your configuration:

```
log4j.logger.org.apache.camel.component.smpp=DEBUG
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

SNMP COMPONENT

Available as of Camel 2.1

The **snmp**: component gives you the ability to poll SNMP capable devices or receiving traps.

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

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

URI format

```
snmp://hostname[:port][?Options]
```

The component supports polling OID values from an SNMP enabled device and receiving traps.

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

Options

Name	Default Value	Description
type	none	The type of action you want to perform. Actually you can enter here POLL or TRAP. The value POLL will instruct the endpoint to poll a given host for the supplied OID keys. If you put in TRAP you will setup a listener for SNMP Trap Events.
protocol	udp	Here you can select which protocol to use. You can use either udp or tcp.
retry	2	Defines how many a retry is made if a response is not received.
timeout	1000	
smpVersion	(which means SNMPv1)	Sets the snmp version for the request.
snmpCommunity	public	Community string for the snmp request.
delay	60 seconds	Defines the delay in seconds between to poll cycles. From Camel 2.15 onwards the delay is using millis as its timeout, so configure 30000 for 30 seconds. Older releases uses the value in seconds.
oids	none	Defines which values you are interested in. Please have a look at the WMI section to get a better understanding. You may provide a single OID or a coma separated list of OIDs. Example: oid1.3.6.1.2.1.1.3.0,oid1.3.6.1.2.1.25.3.2.1.5.1,oid1.3.6.1.2.1.25.3.5.1.1.1,oid1.3.6.1.2.1.43.5.1.1.11.1

The result of a poll

Given the situation, that I poll for the following OIDs:

Listing 1. OIDs

```
1.3.6.1.2.1.1.3.0
1.3.6.1.2.1.25.3.2.1.5.1
1.3.6.1.2.1.25.3.5.1.1.1
1.3.6.1.2.1.43.5.1.1.11.1
```

The result will be the following:

Listing 1. Result of toString conversion

```
<?xml version="1.0" encoding="UTF-8"?>
<snmp>
    <entry>
        <oid>1.3.6.1.2.1.1.3.0</oid>
        <value>6 days, 21:14:28.00</value>
    </entry>
    <entry>
        <oid>1.3.6.1.2.1.25.3.2.1.5.1</oid>
        <value>2</value>
    </entry>
    <entry>
        <oid>1.3.6.1.2.1.25.3.5.1.1.1</oid>
        <value>3</value>
    </entry>
    <entry>
        <oid>1.3.6.1.2.1.43.5.1.1.11.1</oid>
        <value></value>
    </entry>
</snmp>
```

```

<oid>1.3.6.1.2.1.43.5.1.1.11.1</oid>
<value>6</value>
</entry>
<entry>
<oid>1.3.6.1.2.1.1.1.0</oid>
<value>My Very Special Printer Of Brand Unknown</value>
</entry>
</snmp>

```

As you maybe recognized there is one more result than requested....1.3.6.1.2.1.1.1.0.
This one is filled in by the device automatically in this special case. So it may absolutely happen, that you receive more than you requested...be prepared.

Examples

Polling a remote device:

```
snmp:192.168.178.23:161?protocol=udp&type=POLL&oids=1.3.6.1.2.1.1.5.0
```

Setting up a trap receiver (**Note that no OID info is needed here!**):

```
snmp:127.0.0.1:162?protocol=udp&type=TRAP
```

From Camel 2.10.0, you can get the community of SNMP TRAP with message header 'securityName', peer address of the SNMP TRAP with message header 'peerAddress'.

Routing example in Java: (converts the SNMP PDU to XML String)

```
from("snmp:192.168.178.23:161?protocol=udp&type=POLL&oids=1.3.6.1.2.1.1.5.0").
convertBodyTo(String.class).
to("activemq:snmp.states");
```

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.

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

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

URI format

spring-integration:defaultChannelName[?options]

Where **defaultChannelName** represents the default channel name which is used by the Spring Integration Spring context. It will equal to the **inputchannel** name for the Spring Integration consumer and the **outputchannel** name for the Spring Integration provider.

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

Options

confluenceTableSmall

Name	Type	Description
inputchannel	String	The Spring integration input channel name that this endpoint wants to consume from, where the specified channel name is defined in the Spring context.
outputchannel	String	The Spring integration output channel name that is used to send messages to the Spring integration context.
inout	String	The exchange pattern that the Spring integration endpoint should use. If inout=true then a reply channel is expected, either from the Spring Integration Message header or configured on the endpoint.

Usage

The Spring integration component is a bridge that connects Camel endpoints with Spring integration endpoints through the Spring integration's input channels and output channels. Using this component, we can send Camel messages to Spring Integration endpoints or receive messages from Spring integration endpoints in a Camel routing context.

Examples

Using the Spring integration endpoint

You can set up a Spring integration endpoint using a URI, as follows:

```
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring-integration/src/test/resources/org/apache/camel/component/spring/integration/producer.xml}
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring-integration/src/test/resources/org/apache/camel/component/spring/integration/twoWayConsumer.xml}
Or directly using a Spring integration channel name:
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring-integration/src/test/resources/org/apache/camel/component/spring/integration/springChannelConverter.xml}
```

The Source and Target adapter

Spring integration also provides the Spring integration's source and target adapters, which can route messages from a Spring integration channel to a Camel endpoint or from a Camel endpoint to a Spring integration channel.

This example uses the following namespaces:

```
{snippet:id=header|lang=xml|url=camel/trunk/components/camel-spring-integration/src/test/resources/org/apache/camel/component/spring/integration/adapter/CamelTarget.xml}
```

You can bind your source or target to a Camel endpoint as follows:

```
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring-integration/src/test/resources/org/apache/camel/component/spring/integration/adapter/CamelTarget.xml}
{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring-integration/src/test/resources/org/apache/camel/component/spring/integration/adapter/CamelSource.xml}
```

Endpoint See Also

SPRING LDAP COMPONENT

Available since Camel 2.11

The **spring-ldap** component provides a Camel wrapper for Spring LDAP.

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

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

URI format

```
spring-ldap:springLdapTemplate[?options]
```

Where **springLdapTemplate** is the name of the Spring LDAP Template bean. In this bean, you configure the URL and the credentials for your LDAP access.

Options

Name	Type	Description
operation	String	The LDAP operation to be performed. Must be one of search, bind, or unbind.
scope	String	The scope of the search operation. Must be one of object, onelevel, or subtree, see also http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol#Search_and_Compare

If an unsupported value is specified for some option, the component throws an `UnsupportedOperationException`.

Usage

The component supports producer endpoint only. An attempt to create a consumer endpoint will result in an `UnsupportedOperationException`.

The body of the message must be a map (an instance of `java.util.Map`). This map must contain at least an entry with the key `dn` that specifies the root node for the LDAP operation to be performed. Other entries of the map are operation-specific (see below).

The body of the message remains unchanged for the bind and unbind operations. For the search operation, the body is set to the result of the search, see <http://static.springsource.org/spring-ldap/site/apidocs/org/springframework/ldap/core/LdapTemplate.html#search%28java.lang.String,%20java.lang.String,%20int,%20or>

Search

The message body must have an entry with the key `filter`. The value must be a String representing a valid LDAP filter, see http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol#Search_and_Compare.

Bind

The message body must have an entry with the key `attributes`. The value must be an instance of `javax.naming.directory.Attributes` This entry specifies the LDAP node to be created.

Unbind

No further entries necessary, the node with the specified dn is deleted.

Key definitions

In order to avoid spelling errors, the following constants are defined in `org.apache.camel.spring.ldap.SpringLdapProducer`:

- `public static final String DN = "dn"`
- `public static final String FILTER = "filter"`
- `public static final String ATTRIBUTES = "attributes"`

SPRING WEB SERVICES COMPONENT

Available as of Camel 2.6

The `spring-ws`: component allows you to integrate with Spring Web Services. It offers both *client-side* support, for accessing web services, and *server-side* support for creating your own contract-first web services.

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

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

Dependencies

As of Camel 2.8 this component ships with Spring-WS 2.0.x which (like the rest of Camel) requires Spring 3.0.x.

Earlier Camel versions shipped Spring-WS 1.5.9 which is compatible with Spring 2.5.x and 3.0.x. In order to run earlier versions of `camel-spring-ws` on Spring 2.5.x you need to add the `spring-webmvc` module from Spring 2.5.x. In order to run Spring-WS 1.5.9 on Spring 3.0.x you need to exclude the OXM module from Spring 3.0.x as this module is also included in Spring-WS 1.5.9 (see this post)

URI format

The URI scheme for this component is as follows

```
spring-ws:[mapping-type:]address[?options]
```

To expose a web service `mapping-type` needs to be set to any of the following:

Mapping type	Description
<code>rootname</code>	Offers the option to map web service requests based on the qualified name of the root element contained in the message.
<code>soapaction</code>	Used to map web service requests based on the SOAP action specified in the header of the message.
<code>uri</code>	In order to map web service requests that target a specific URI.
<code>xpathresult</code>	Used to map web service requests based on the evaluation of an XPath expression against the incoming message. The result of the evaluation should match the XPath result specified in the endpoint URI.
<code>beanname</code>	Allows you to reference an <code>org.apache.camel.component.spring.ws.bean.CamelEndpointSoapMapper</code> object in order to integrate with existing (legacy) endpoint mappings like <code>PayloadToAddressNameEndpointMapping</code> .
<code>soapActionInMapping</code>	

As a consumer the `address` should contain a value relevant to the specified mapping-type (e.g. a SOAP action, XPath expression). As a producer the address should be set to the URI of the web service your calling upon.

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

Options

Name	Required?	Description
<code>soapaction</code>	No	SOAP action to include inside a SOAP request when accessing remote web services
<code>wsaddressingAction</code>	No	WS-Addressing 1.0 action header to include when accessing web services. The <code>To</code> header is set to the address of the web service as specified in the endpoint URI (default Spring-WS behavior).
<code>outputAction</code>	No	Signifies the value for the response WS-Addressing Action header that is provided by the method.
<code>faultAction</code>	No	Signifies the value for the faultAction response WS-Addressing Fault Action header that is provided by the method.
<code>faultTo</code>	No	Signifies the value for the faultTo response WS-Addressing FaultTo header that is provided by the method.
<code>replyTo</code>	No	Signifies the value for the replyTo response WS-Addressing ReplyTo header that is provided by the method.
<code>expression</code>	Only when <code>mapping-type=xpathresult</code>	XPath expression to use in the process of mapping web service requests, should match the result specified by <code>xpathresult</code>
<code>timeout</code>	No	Camel 2.10: Sets the socket read timeout (in milliseconds) while invoking a webservices using the producer, see <code>URLConnection.setReadTimeout()</code> and <code>URLConnection.setConnectTimeout()</code>

<small>CommonsHttpMessageSender.setReadTimeout()}. This option works when using the built-in message sender implementations: <code>CommonsHttpMessageSender</code> and <code>HttpURLConnectionMessageSender</code>. One of these implementations will be used by default for HTTP based services unless you customize the Spring WS configuration options supplied to the component. If you are using a non-standard sender, it is assumed that you will handle your own timeout configuration.</small>		
<small>Camel 2.12.0: The built-in message sender <code>HttpComponentsMessageSender</code> is considered instead of <code>CommonsHttpMessageSender</code> which has been deprecated.</small>		
<small>Camel 2.10: Reference to an org.apache.camel.util.jsse.SSLEnvironmentParameters in the Registry. See Using the JSSE Configuration Utility. This option works when using the built-in message sender implementations: <code>CommonsHttpMessageSender</code> and <code>HttpURLConnectionMessageSender</code>. One of these implementations will be used by default for HTTP based services unless you customize the Spring WS configuration options supplied to the component. If you are using a non-standard sender, it is assumed that you will handle your own TLS configuration.</small>		
<small>Camel 2.12: The built-in message sender <code>HttpComponentsMessageSender</code> is considered instead of <code>CommonsHttpMessageSender</code> which has been deprecated.</small>		
<small>Camel 2.10: Reference to an org.apache.camel.component.spring.ws.WebServiceTemplate in the Registry. This allows for full control over client-side web services handling, like adding a custom interceptor or specifying a fault resolver, message sender or message factory.</small>		
<small>messageSender No Option to provide a custom WebServiceMessageSender. For example to perform authentication or use alternative transports</small>		
<small>messageFactory No Option to provide a custom WebServiceMessageFactory. For example when you want Apache Axiom to handle web service messages instead of SAAJ</small>		
<small>endpointsMappingKey No Reference to an org.apache.camel.component.spring.ws.EndpointMapping by name</small>		
<small>endpointsMapping Only when mapping-type is rootaction, soapaction-uri or endpoint-uri Reference to an instance of org.apache.camel.component.spring.ws.bean.CamelEndpointMapping in the Registry/ApplicationContext. Only one bean is required in the registry to serve all endpoints. Camel endpoints can be auto-discovered by the MessageDispatcher and used to map requests to Camel endpoints based on characteristics specified on the endpoint (like root QName, SOAP action, etc)</small>		
<small>endpointDispatcher No Spring Integration endpoint dispatcher for server endpoint. MessageDispatcher for dispatching messages received by Spring-WS to a Camel endpoint, to integrate with existing (legacy) endpoint mappings like <code>PayloadRootNameEndpointMapping</code>, <code>SoapActionEndpointMapping</code>, etc</small>		
<small>messageFilter No Camel 2.10.3 Option to provide a custom MessageFilter. For example when you want to process your headers or attachments by your own.</small>		
<small>messageProcessor No A Camel Processor to process the message before it is sent to the endpoint</small>		
<small>webServiceEndpointUri No The default Web Service endpoint uri to use for the producer</small>		

Message headers

Name	Type	Description
CamelSpringWebServiceEndpointUri	String	URI of the web service you're accessing as a client, overrides address part of the endpoint URI
CamelSpringWebServiceSoapAction	String	Header to specify the SOAP action of the message, overrides soapAction option if present
CamelSpringWebServiceSoapHeader	Source	Camel 2.11.1: Use this header to specify/access the SOAP headers of the message
CamelSpringWebServiceAddressingInAction	URI	Use this header to specify the WS-Addressing action of the message, overrides wsAddressingInAction option if present
CamelSpringWebServiceAddressingFaultTo	URI	Use this header to specify the WS-Addressing FaultTo, overrides faultTo option if present
CamelSpringWebServiceAddressingReplyTo	URI	Use this header to specify the WS-Addressing ReplyTo, overrides replyTo option if present
CamelSpringWebServiceAddressingOutAction	URI	Use this header to specify the WS-Addressing Action, overrides outAction option if present
CamelSpringWebServiceAddressingFaultAction	URI	Use this header to specify the WS-Addressing Fault Action, overrides faultAction option if present

ACCESSING WEB SERVICES

To call a web service at `http://foo.com/bar` simply define a route:

```
from("direct:example").to("spring-ws:http://foo.com/bar")
```

And sent a message:

```
template.requestBody("direct:example", "<foobar xmlns='http://foo.com'><msg>test message</msg></foobar>");
```

Remember if it's a SOAP service you're calling you don't have to include SOAP tags. Spring-WS will perform the XML-to-SOAP marshaling.

Sending SOAP and WS-Addressing action headers

When a remote web service requires a SOAP action or use of the WS-Addressing standard you define your route as:

```
from("direct:example")
.to("spring-ws:http://foo.com/bar?soapAction=http://foo.com&wsAddressingAction=http://bar.com")
```

Optionally you can override the endpoint options with header values:

```
template.requestBodyAndHeader("direct:example",
"<foobar xmlns='http://foo.com'><msg>test message</msg></foobar>",
SpringWebserviceConstants.SPRING_WS_SOAP_ACTION, "http://baz.com");
```

Using SOAP headers

Available as of Camel 2.11.1

You can provide the SOAP header(s) as a Camel Message header when sending a message to a spring-ws endpoint, for example given the following SOAP header in a String

```
String body = ...
String soapHeader = "<h:Header xmlns:h='http://www.webservices.NET/'><h:MessageID>1234567890</h:MessageID><h:Nested><h:NestedID>1111</h:Nest
```

We can set the body and header on the Camel Message as follows:

```
exchange.getIn().setBody(body);
exchange.getIn().setHeader(SpringWebserviceConstants.SPRING_WS_SOAP_HEADER, soapHeader);
```

And then send the Exchange to a spring-ws endpoint to call the Web Service.

Likewise the spring-ws consumer will also enrich the Camel Message with the SOAP header.

For an example see this unit test.

The header and attachment propagation

Spring WS Camel supports propagation of the headers and attachments into Spring-WS WebServiceMessage response since version 2.10.3. The endpoint will use so called "hook" the MessageFilter (default implementation is provided by BasicMessageFilter) to propagate the exchange headers and attachments into WebServiceMessage response. Now you can use

```
exchange.getOut().getHeaders().put("myCustom", "myHeaderValue")
exchange.getIn().addAttachment("myAttachment", new DataHandler(...))
```

Note: If the exchange header in the pipeline contains text, it generates QName(key)=value attribute in the soap header. Recommended is to create a QName class directly and put into any key into header.

How to use MTOM attachments

The BasicMessageFilter provides all required information for Apache Axiom in order to produce MTOM message. If you want to use Apache Camel Spring WS within Apache Axiom, here is an example:

- Simply define the messageFactory as is bellow and Spring-WS will use MTOM strategy to populate your SOAP message with optimized attachments.

```
<bean id="axiomMessageFactory"
  class="org.springframework.ws.soap.axiom.AxiomSoapMessageFactory">
<property name="payloadCaching" value="false" />
<property name="attachmentCaching" value="true" />
<property name="attachmentCacheThreshold" value="1024" />
</bean>
```

- Add into your pom.xml the following dependencies

```
<dependency>
<groupId>org.apache.ws.commons.axiom</groupId>
<artifactId>axiom-api</artifactId>
<version>1.2.13</version>
</dependency>
<dependency>
<groupId>org.apache.ws.commons.axiom</groupId>
<artifactId>axiom-impl</artifactId>
<version>1.2.13</version>
<scope>runtime</scope>
</dependency>
```

- Add your attachment into the pipeline, for example using a Processor implementation.

```
private class Attachment implements Processor {
    public void process(Exchange exchange) throws Exception
    { exchange.getOut().copyFrom(exchange.getIn()); File file = new File("testAttachment.txt"); exchange.getOut().addAttachment("test", new DataHandler(file));
    }
```

- Define endpoint (producer) as usual, for example like this:

```
from("direct:send")
.process(new Attachment())
.to("spring-ws:http://localhost:8089/mySoapService?soapAction=mySoap&messageFactory=axiomMessageFactory");
```

- Now, your producer will generate MTOM message with otpimized attachments.

The custom header and attachment filtering

If you need to provide your custom processing of either headers or attachments, extend existing BasicMessageFilter and override the appropriate methods or write a brand new implementation of the MessageFilter interface.

To use your custom filter, add this into your spring context:

You can specify either a global a or a local message filter as follows:

- the global custom filter that provides the global configuration for all Spring-WS endpoints

```
<bean id="messagefilter" class="your.domain.myMessageFilter" scope="singleton" />
```

or

- the local messageFilter directly on the endpoint as follows:

```
to("spring-ws:http://yourdomain.com?messageFilter=#myEndpointSpecificMessageFilter");
```

For more information see CAMEL-5724

If you want to create your own MessageFilter, consider overriding the following methods in the default implementation of MessageFilter in class BasicMessageFilter:

```
protected void doProcessSoapHeader(Message inOrOut, SoapMessage soapMessage)
{your code /*no need to call super*/ }
```

```
protected void doProcessSoapAttachments(Message inOrOut, SoapMessage response)
{ your code /*no need to call super*/ }
```

Using a custom MessageSender and MessageFactory

A custom message sender or factory in the registry can be referenced like this:

```
from("direct:example")
.to("spring-ws:http://foo.com/bar?messageFactory=#messageFactory&messageSender=#messageSender")
```

Spring configuration:

```
<!-- authenticate using HTTP Basic Authentication -->
<bean id="messageSender" class="org.springframework.ws.transport.http.HttpComponentsMessageSender">
    <property name="credentials">
        <bean class="org.apache.commons.httpclient.UsernamePasswordCredentials">
            <constructor-arg index="0" value="admin"/>
            <constructor-arg index="1" value="secret"/>
        </bean>
    </property>
</bean>

<!-- force use of Sun SAAJ implementation, http://static.springsource.org/spring-ws/sites/1.5/faq.html#saj-jboss -->
<bean id="messageFactory" class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory">
    <property name="messageFactory">
        <bean class="com.sun.xml.messaging.saaj.soap.ver1_1.SOAPMessageFactory1_1Impl"/>
    </property>
</bean>
```

EXPOSING WEB SERVICES

In order to expose a web service using this component you first need to set-up a MessageDispatcher to look for endpoint mappings in a Spring XML file. If you plan on running inside a servlet container you probably want to use a `MessageDispatcherServlet` configured in `web.xml`.

By default the `MessageDispatcherServlet` will look for a Spring XML named `/WEB-INF/spring-ws-servlet.xml`. To use Camel with Spring-WS the only mandatory bean in that XML file is `CamelEndpointMapping`. This bean allows the `MessageDispatcher` to dispatch web service requests to your routes.

`web.xml`

```
<web-app>
    <servlet>
        <servlet-name>spring-ws</servlet-name>
        <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

`spring-ws-servlet.xml`

```
<bean id="endpointMapping" class="org.apache.camel.component.spring.ws.bean.CamelEndpointMapping" />

<bean id="wsdl1" class="org.springframework.ws.wsdl11.DefaultWsdl11Definition">
    <property name="schema">
        <bean class="org.springframework.xml.xsd.SimpleXsdSchema">
            <property name="xsd" value="/WEB-INF/foobar.xsd"/>
        </bean>
    </property>
    <property name="portTypeName" value="FooBar"/>
    <property name="locationUri" value="/" />
    <property name="targetNamespace" value="http://example.com/" />
</bean>
```

More information on setting up Spring-WS can be found in [Writing Contract-First Web Services](#). Basically paragraph 3.6 "Implementing the Endpoint" is handled by this component (specifically paragraph 3.6.2 "Routing the Message to the Endpoint" is where `CamelEndpointMapping` comes in). Also don't forget to check out the [Spring Web Services Example](#) included in the Camel distribution.

Endpoint mapping in routes

With the XML configuration in-place you can now use Camel's DSL to define what web service requests are handled by your endpoint:

The following route will receive all web service requests that have a root element named "GetFoo" within the `http://example.com/` namespace.

```
from("spring-ws:root QName:{http://example.com/}GetFoo?endpointMapping=#endpointMapping")
    .convertBodyTo(String.class).to(mock:example)
```

The following route will receive web service requests containing the `http://example.com/GetFoo` SOAP action.

```
from("spring-ws:soapaction:http://example.com/GetFoo?endpointMapping=#endpointMapping")
.convertBodyToString(String.class).to(mock:example)
```

The following route will receive all requests sent to `http://example.com/foobar`.

```
from("spring-ws:uri:http://example.com/foobar?endpointMapping=#endpointMapping")
.convertBodyToString(String.class).to(mock:example)
```

The route below will receive requests that contain the element `<foobar>abc</foobar>` anywhere inside the message (and the default namespace).

```
from("spring-ws:xpathresult:abc?expression=/foobar&endpointMapping=#endpointMapping")
.convertBodyToString(String.class).to(mock:example)
```

Alternative configuration, using existing endpoint mappings

For every endpoint with mapping-type beanname one bean of type `CamelEndpointDispatcher` with a corresponding name is required in the Registry/ApplicationContext. This bean acts as a bridge between the Camel endpoint and an existing endpoint mapping like `PayloadRoot QName EndpointMapping`.

The use of the beanname mapping-type is primarily meant for (legacy) situations where you're already using Spring-WS and have endpoint mappings defined in a Spring XML file. The beanname mapping-type allows you to wire your Camel route into an existing endpoint mapping. When you're starting from scratch it's recommended to define your endpoint mappings as Camel URI's (as illustrated above with `endpointMapping`) since it requires less configuration and is more expressive. Alternatively you could use vanilla Spring-WS with the help of annotations.

An example of a route using beanname:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="spring-ws:beanname:QuoteEndpointDispatcher" />
    <to uri="mock:example" />
  </route>
</camelContext>

<bean id="legacyEndpointMapping" class="org.springframework.ws.server.endpoint.mapping.PayloadRoot QName EndpointMapping">
  <property name="mappings">
    <props>
      <prop key="http://example.com/GetFuture">FutureEndpointDispatcher</prop>
      <prop key="http://example.com/GetQuote">QuoteEndpointDispatcher</prop>
    </props>
  </property>
</bean>

<bean id="QuoteEndpointDispatcher" class="org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher" />
<bean id="FutureEndpointDispatcher" class="org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher" />
```

POJO (UN)MARSHALLING

Camel's pluggable data formats offer support for pojo/xml marshalling using libraries such as JAXB, XStream, JibX, Castor and XMLBeans. You can use these data formats in your route to sent and receive pojo's, to and from web services.

When accessing web services you can marshal the request and unmarshal the response message:

```
JaxbDataFormat jaxb = new JaxbDataFormat(false);
jaxb.setContextPath("com.example.model");

from("direct:example").marshal(jaxb).to("spring-ws:http://foo.com/bar").unmarshal(jaxb);
```

Similarly when providing web services, you can unmarshal XML requests to POJO's and marshal the response message back to XML:

```
from("spring-ws:rootqname:{http://example.com/}GetFoo?endpointMapping=#endpointMapping").unmarshal(jaxb)
.to("mock:example").marshal(jaxb);
```

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 as well as allowing streaming of file and URL.

Maven users will need to add the following dependency to their `pom.xml` for this component:
`<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-stream</artifactId> <version>x.x.x</version> <!-- use the same version as your Camel core version --> </dependency>`

URI format

`stream:[in]?[options] stream:[out]?[options] stream:[err]?[options] stream:[header]?[options]`

In addition, the `file` and `url` endpoint URLs are supported:

`stream:[file?fileName=/foo/bar.txt] stream:[url]?[options]`

If the `stream:[header]` URI is specified, the `stream.header` is used to find the stream to write to. This option is available only for stream producers (that is, it cannot appear in `from()`).

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

Options

confluenceTableSmall

Name	Default Value	Description
<code>delay</code>	0	Initial delay in milliseconds before consuming or producing the stream.
<code>encoding</code>	<code>JVM Default</code>	You can configure the encoding (is a charset name) to use text-based streams (for example, message body is a <code>String</code> object). If not provided, Camel uses the JVM default Charset.
<code>promptMessage</code>	<code>null</code>	Message prompt to use when reading from <code>stream:[in]</code> ; for example, you could set this to <code>Enter a command:</code>
<code>promptDelay</code>	0	Optional delay in milliseconds before showing the message prompt.
<code>initialPromptDelay</code>	2000	Initial delay in milliseconds before showing the message prompt. This delay occurs only once. Can be used during system startup to avoid message prompts being written while other logging is done to the system out.
<code>fileName</code>	<code>null</code>	When using the <code>stream:[file]</code> URI format, this option specifies the filename to stream to/from.
<code>url</code>	<code>null</code>	When using the <code>stream:[url]</code> URI format, this option specifies the URL to stream to/from. The input/output stream will be opened using the JDK <code>URLConnection</code> facility.
<code>scanStream</code>	<code>false</code>	To be used for continuously reading a stream such as the unix <code>tail</code> command. Camel 2.4 to Camel 2.6: will retry opening the file if it is overwritten, somewhat like <code>tail --retry</code>
<code>scanStreamDelay</code>	0	Delay in milliseconds between read attempts when using <code>scanStream</code> .
<code>groupLines</code>	0	Camel 2.5: To group X number of lines in the consumer. For example to group 10 lines and therefore only spit out an Exchange with 10 lines, instead of 1 Exchange per line.
<code>autoCloseCount</code>	0	Camel 2.10.0: (2.9.3 and 2.8.6) Number of messages to process before closing stream on Producer side. Never close stream by default (only when Producer is stopped). If more messages are sent, the stream is reopened for another <code>autoCloseCount</code> batch.
<code>closeOnDone</code>	<code>false</code>	Camel 2.11.0: This option is used in combination with Splitter and streaming to the same file. The idea is to keep the stream open and only close when the Splitter is done, to improve performance. Mind this requires that you only stream to the same file, and not 2 or more files, and that the last split message that carries the information that its the last, is routed to the stream endpoint so it gets the signal to close.

Message content

The **stream:** component supports either `String` or `byte[]` for writing to streams. Just add either `String` or `byte[]` content to the `message.in.body`. Messages sent to the **stream:** producer in binary mode are not followed by the newline character (as opposed to the `String` messages). Message with `null` body will not be appended to the output stream.

The special `stream:[header]` URI is used for custom output streams. Just add a `java.io.OutputStream` object to `message.in.header` in the key header.

See samples for an example.

Samples

In the following sample we route messages from the `direct:in` endpoint to the `System.out` stream:

```
java// Route messages to the standard output. from("direct:in").to("stream:out"); // Send String payload to the standard output. // Message will be followed by the newline. template.sendBody("direct:in", "Hello Text World"); // Send byte[] payload to the standard output. // No newline will be added after the message. template.sendBody("direct:in", "Hello Bytes World".getBytes()); The following sample demonstrates how the header type can be used to determine which stream to use. In the sample we use our own output stream, MyoutputStream.{snippet:id=e1|lang=java|url=camel/trunk/components/camel-stream/src/test/java/org/apache/camel/component/stream/StreamHeaderTest.java}The following sample demonstrates how to continuously read a file stream (analogous to the UNIX tail command): javafrom("stream:file?fileName=/server/logs/server.log&scanStream=true&scanStreamDelay=1000").to("bean:logService?method=parseLogLine"); One gotcha with scanStream (pre Camel 2.7) or scanStream + retry is the file will be re-opened and scanned with each iteration of scanStreamDelay. Until NIO2 is available we cannot reliably detect when a file is deleted/recreated.
```

Endpoint See Also

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.

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

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

URI Format

`string-template:templateName[?options]`

Where `templateName` is the classpath-local URI of the template to invoke; or the complete URL of the remote template.

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

Options

confluenceTableSmall

Option	Default	Description
<code>contentCache</code>	false	Cache for the resource content when its loaded. Note: as of Camel 2.9 cached resource content can be cleared via JMX using the endpoint's <code>clearContentCache</code> operation.
<code>delimiterStart</code>	null	From Camel 2.11.1 : configuring the variable start delimiter
<code>delimiterStop</code>	null	From Camel 2.11.1 : configuring the variable end delimiter

Headers

Camel will store a reference to the resource in the message header with key, `org.apache.camel.stringtemplate.resource`. The Resource is an `org.springframework.core.io.Resource` object.

Hot-Reloading

The string template resource is by default hot-reloadable for both file and classpath resources (expanded jar). If you set `contentCache=true`, Camel will load the resource just once, disabling hot-reloading. This scenario can be used in production when the resource never changes.

StringTemplate Attributes

Camel will provide exchange information as attributes (just a `java.util.Map`) to the string template. The Exchange is transferred as:

confluenceTableSmall

Key	Value
<code>exchange</code>	The Exchange itself.
<code>headers</code>	The headers of the IN message.
<code>camelContext</code>	The Camel Context.
<code>request</code>	The IN message.
<code>in</code>	The IN message.
<code>body</code>	The IN message body.
<code>out</code>	The OUT message (only for <code>Inout</code> message exchange pattern).
<code>response</code>	

The OUT message (only for Inout message exchange pattern).

From **Camel 2.14**: you can define the custom context map by setting the message header CamelStringTemplatevariableMap, as shown below:

```
javaMap<String, Object> variableMap = new HashMap<String, Object>(); Map<String, Object> headersMap = new  
HashMap<String, Object>(); headersMap.put("name", "Willem"); variableMap.put("headers", headersMap);  
variableMap.put("body", "Monday"); variableMap.put("exchange", exchange);  
exchange.getIn().setHeader("CamelStringTemplateVariableMap", variableMap);
```

Samples

For example you could use a string template as follows in order to formulate a response to a message:
from("activemq:My.Queue") .to("string-template:com/acme/MyResponse.tm");

The Email Sample

In this sample we want to use a string template to send an order confirmation email. The email template is laid out in StringTemplate as:

This example works for **camel 2.11.0**. If your camel version is less than **2.11.0**, the variables should be started and ended with \$. Dear <headers.lastName>, <headers.firstName> Thanks for the order of <headers.item>. Regards Camel Riders Bookstore </body>

And the java code is as follows:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-stringtemplate/src/test/java/org/apache/camel/component/stringtemplate/StringTemplateLetterTest.java}Endpoint See Also

SQL COMPONENT

The **sql**: component allows you to work with databases using JDBC queries. The difference between this component and JDBC component is that in case of SQL the query is a property of the endpoint and it uses message payload as parameters passed to the query.

This component uses **spring-jdbc** behind the scenes for the actual SQL handling.

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

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

The SQL component also supports:

- a JDBC based repository for the Idempotent Consumer EIP pattern. See further below.
- a JDBC based repository for the Aggregator EIP pattern. See further below.

URI format

From Camel 2.11 onwards this component can create both consumer (e.g. from()) and producer endpoints (e.g. to()).

In previous versions, it could only act as a producer.

This component can be used as a Transactional Client.

The SQL component uses the following endpoint URI notation:

```
sql:select * from table where id=# order by name[?options]
```

From Camel 2.11 onwards you can use named parameters by using :#name_of_the_parameter style as shown:

```
sql:select * from table where id=:myId order by name[?options]
```

When using named parameters, Camel will lookup the names from, in the given precedence:

1. from message body if its a java.util.Map

2. from message headers

If a named parameter cannot be resolved, then an exception is thrown.

From **Camel 2.14** onward you can use Simple expressions as parameters as shown:

```
sql:select * from table where id:=${property.myId} order by name[?options]
```

Notice that the standard ? symbol that denotes the parameters to an SQL query is substituted with the # symbol, because the ? symbol is used to specify options for the endpoint. The ? symbol replacement can be configured on endpoint basis.

From **Camel 2.17** onwards you can externalize your SQL queries to files in the classpath or file system as shown:

```
sql:classpath:sql/myquery.sql[?options]
```

And the myquery.sql file is in the classpath and is just a plain text

```
-- this is a comment select * from table where id = :#${property.myId} order by name
```

In the file you can use multilines and format the SQL as you wish. And also use comments such as the – dash line.

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

Options

confluenceTableSmall

Option	Type	Default	Description
--------	------	---------	-------------

batch	boolean	false	Camel 2.7.5, 2.8.4 and 2.9: Execute SQL batch update statements. See notes below on how the treatment of the inbound message body changes if this is set to true.
dataSourceRef	String	null	- Deprecated and will be removed in Camel 3.0: Reference to a DataSource to look up in the registry. Use dataSource=#theName instead.
dataSource	String	null	Camel 2.11: Reference to a DataSource to look up in the registry.
placeholder	String	#	Camel 2.4: Specifies a character that will be replaced to ? in SQL query. Notice, that it is simple String.replaceAll() operation and no SQL parsing is involved (quoted strings will also change). This replacement is only happening if the endpoint is created using the sqlComponent. If you manually create the endpoint, then use the expected ? sign instead.
usePlaceholder	boolean	true	Camel 2.17: Sets whether to use placeholder and replace all placeholder characters with ? sign in the SQL queries.
template.<xxx>		null	Sets additional options on the Spring JdbcTemplate that is used behind the scenes to execute the queries. For instance, template.setMaxRows=10. For detailed documentation, see the JdbcTemplate javadoc documentation.
allowNamedParameters	boolean	true	Camel 2.11: Whether to allow using named parameters in the queries.
processingStrategy			Camel 2.11: SQL consumer only: Allows to plugin to use a custom org.apache.camel.component.sql.SqlProcessingStrategy to execute queries when the consumer has processed the rows/batch.
prepareStatementStrategy			Camel 2.11: Allows to plugin to use a custom org.apache.camel.component.sql.SqlPrepareStatementStrategy to control preparation of the query and prepared statement.
consumer.delay	long	500	Camel 2.11: SQL consumer only: Delay in milliseconds between each poll.
consumer.initialDelay	long	1000	Camel 2.11: SQL consumer only: Milliseconds before polling starts.
consumer.useFixedDelay	boolean	false	Camel 2.11: SQL consumer only: Set to true to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
maxMessagesPerPoll	int	0	Camel 2.11: SQL consumer only: An integer value to define the maximum number of messages to gather per poll. By default, no maximum is set.
useIterator	boolean	true	Camel 2.11: SQL consumer only: If true each row returned when polling will be processed individually. If false the entire java.util.List of data is set as the IN body. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
routeEmptyResultSet	boolean	false	Camel 2.11: SQL consumer only: Whether to route a single empty Exchange if there was no data to poll. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
onConsume	String	null	Camel 2.11: SQL consumer only: After processing each row then this query can be executed, if the Exchange was processed successfully, for example to mark the row as processed. The query can have parameter. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
onConsumeFailed	String	null	Camel 2.11: SQL consumer only: After processing each row then this query can be executed, if the Exchange failed, for example to mark the row as failed. The query can have parameter. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
onConsumeBatchComplete	String	null	Camel 2.11: SQL consumer only: After processing the entire batch, this query can be executed to bulk update rows etc. The query cannot have parameters. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
expectedupdateCount	int	-1	Camel 2.11: SQL consumer only: If using consumer.onConsume then this option can be used to set an expected number of rows being updated. Typically you may set this to 1 to expect one row to be updated. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
breakBatchOnConsumeFail	boolean	false	Camel 2.11: SQL consumer only: If using consumer.onConsume and

			it fails, then this option controls whether to break out of the batch or continue processing the next row from the batch. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
alwaysPopulateStatement	boolean	false	Camel 2.11: SQL producer only: If enabled then the populateStatement method from org.apache.camel.component.sql.SqlPreparedStatementStrategy is always invoked, also if there is no expected parameters to be prepared. When this is false then the populateStatement is only invoked if there is 1 or more expected parameters to be set; for example this avoids reading the message body/headers for SQL queries with no parameters.
separator	char	,	Camel 2.11.1: The separator to use when parameter values is taken from message body (if the body is a String type), to be inserted at # placeholders. Notice if you use named parameters, then a Map type is used instead.
outputType	String	SelectList	Camel 2.12.0: outputType='SelectList', for consumer or producer, will output a List of Map. SelectOne will output single Java object in the following way: a) If the query has only single column, then that JDBC Column object is returned. (such as SELECT COUNT(*) FROM PROJECT will return a Long object. b) If the query has more than one column, then it will return a Map of that result. c) If the outputClass is set, then it will convert the query result into an Java bean object by calling all the setters that match the column names. It will assume your class has a default constructor to create instance with. d) If the query resulted in more than one rows, it throws an non-unique result exception. From Camel 2.14.1 onwards the SelectList also supports mapping each row to a Java object as the SelectOne does (only step c). From Camel 2.18 onwards there is a new StreamList outputType that streams the result of the query using an Iterator. It can be used with the Splitter EIP in streaming mode to process the ResultSet in streaming fashion. This StreamList do not support batch mode, but you can use outputClass to map each row to a class.
outputClass	String	null	Camel 2.12.0: Specify the full package and class name to use as conversion when outputType=SelectOne.
outputHeader	String	null	Camel 2.15: To store the result as a header instead of the message body. This allows to preserve the existing message body as-is.
parametersCount	int	0	Camel 2.11.2/2.12.0 If set greater than zero, then Camel will use this count value of parameters to replace instead of querying via JDBC metadata API. This is useful if the JDBC vendor could not return correct parameters count, then user may override instead.
noop	boolean	false	Camel 2.12.0 If set, will ignore the results of the SQL query and use the existing IN message as the OUT message for the continuation of processing
useMessageBodyForSql	boolean	false	Camel 2.16: Whether to use the message body as the SQL and then headers for parameters. If this option is enabled then the SQL in the uri is not used. The SQL parameters must then be provided in a header with the key CamelSqlParameters. This option is only for the producer.
transacted	boolean	false	Camel 2.16.2: SQL consumer only: Enables or disables transaction. If enabled then if processing an exchange failed then the consumer break out processing any further exchanges to cause a rollback eager

Treatment of the message body

The SQL component tries to convert the message body to an object of `java.util.Iterator` type and then uses this iterator to fill the query parameters (where each query parameter is represented by a # symbol (or configured placeholder) in the endpoint URI). If the message body is not an array or collection, the conversion results in an iterator that iterates over only one object, which is the body itself.

For example, if the message body is an instance of `java.util.List`, the first item in the list is substituted into the first occurrence of # in the SQL query, the second item in the list is substituted into the second occurrence of #, and so on.

If batch is set to true, then the interpretation of the inbound message body changes slightly – instead of an iterator of parameters, the component expects an iterator that contains the parameter iterators; the size of the outer iterator determines the batch size.

From Camel 2.16 onwards you can use the option `useMessageBodyForSql` that allows to use the message body as the SQL statement, and then the SQL parameters must be provided in a header with the key `SqIConstants.SQL_PARAMETERS`. This allows the SQL component to work more dynamic as the SQL query is from the message body.

Result of the query

For select operations, the result is an instance of `List<Map<String, Object>>` type, as returned by the `JdbcTemplate.queryForList()` method. For update operations, the result is the number of updated rows, returned as an `Integer`.

By default, the result is placed in the message body. If the `outputHeader` parameter is set, the result is placed in the header. This is an alternative to using a full message enrichment pattern to add headers, it provides a concise syntax for querying a sequence or some other small value into a header. It is convenient to use `outputHeader` and `outputType` together:

```
javafrom("jms:order.inbox") .to("sql:select order_seq.nextval from dual?outputHeader=OrderId&outputType=SelectOne") .to("jms:order.booking");
```

Using StreamList

From **Camel 2.18** onwards the producer supports `outputType=StreamList` that uses an iterator to stream the output of the query. This allows to process the data in a streaming fashion which for example can be used by the Splitter EIP to process each row one at a time, and load data from the database as needed.

```
javafrom("direct:withSplitModel") .to("sql:select * from projects order by id?outputType=StreamList&outputClass=org.apache.camel.component.sql.ProjectModel") .to("log:stream") .split(body()).streaming() .to("log:row") .to("mock:result") .end();
```

Header values

When performing update operations, the SQL Component stores the update count in the following message headers:

Header	Description
<code>CamelSqlUpdateCount</code>	The number of rows updated for update operations, returned as an <code>Integer</code> object. This header is not provided when using <code>outputType=StreamList</code> .
<code>CamelSqlRowCount</code>	The number of rows returned for <code>select</code> operations, returned as an <code>Integer</code> object. This header is not provided when using <code>outputType=StreamList</code> .
<code>CamelSqlQuery</code>	Camel 2.8: Query to execute. This query takes precedence over the query specified in the endpoint URI. Note that query parameters in the header are represented by a ? instead of a # symbol

When performing insert operations, the SQL Component stores the rows with the generated keys and number of these row in the following message headers (**Available as of Camel 2.12.4, 2.13.1:**)

Header	Description
<code>CamelSqlGeneratedKeysRowCount</code>	The number of rows in the header that contains generated keys.
<code>CamelSqlGeneratedKeyRows</code>	Rows that contains the generated keys (a list of maps of keys).

Generated keys

Available as of Camel 2.12.4, 2.13.1 and 2.14

If you insert data using SQL `INSERT`, then the RDBMS may support auto generated keys. You can instruct the SQL producer to return the generated keys in headers.

To do that set the header `CamelSqlRetrieveGeneratedKeys=true`. Then the generated keys will be provided as headers with the keys listed in the table above.

You can see more details in this unit test.

Configuration

You can now set a reference to a `DataSource` in the URI directly:

```
sql:select * from table where id## order by name?dataSource=myDS
```

Sample

In the sample below we execute a query and retrieve the result as a `List` of rows, where each row is a `Map<String, Object>` and the key is the column name.

First, we set up a table to use for our sample. As this is based on an unit test, we do it in java:

```
{snippet:id=e2|lang=java|url=camel/trunk/components/camel-sql/src/test/java/org/apache/camel/component/sql/SqlDataSourceRefTest.java}The SQL script createAndPopulateDatabase.sql we execute looks like as described below:{snippet:id=e1|lang=sql|url=camel/trunk/components/camel-sql/src/test/resources/sql/createAndPopulateDatabase.sql}Then we configure our route and our sql component. Notice that we use a direct endpoint in front of the sql endpoint. This allows us to
```

send an exchange to the `direct` endpoint with the URI, `direct:simple`, which is much easier for the client to use than the long `sql1`: URI. Note that the `DataSource` is looked up in the registry, so we can use standard Spring XML to configure our `DataSource`.{snippet:id=e1|lang=java|url=camel/trunk/components/camel-sql/src/test/java/org/apache/camel/component/sql/SqlDataSourceRefTest.java}And then we fire the message into the `direct` endpoint that will route it to our `sql1` component that queries the database.{snippet:id=e3|lang=java|url=camel/trunk/components/camel-sql/src/test/java/org/apache/camel/component/sql/SqlDataSourceRefTest.java}We could configure the `DataSource` in Spring XML as follows:

```
xml <jee:jndi-lookup id="myDS" jndi-name="jdbc/myDataSource"/>
```

Using named parameters

Available as of Camel 2.11

In the given route below, we want to get all the projects from the projects table. Notice the SQL query has 2 named parameters, `#lic` and `#min`. Camel will then lookup for these parameters from the message body or message headers. Notice in the example above we set two headers with constant value for the named parameters:

```
java from("direct:projects") .setHeader("lic", constant("ASF")) .setHeader("min", constant(123)) .to("sql:select * from projects where license = :#lic and id > :#min order by id")
```

Though if the message body is a `java.util.Map` then the named parameters will be taken from the body.

```
java from("direct:projects") .to("sql:select * from projects where license = :#lic and id > :#min order by id")
```

Using expression parameters

Available as of Camel 2.14

In the given route below, we want to get all the project from the database. It uses the body of the exchange for defining the license and uses the value of a property as the second parameter.

```
java from("direct:projects") .setBody(constant("ASF")) .setProperty("min", constant(123)) .to("sql:select * from projects where license = :#${body} and id > :#${property:min} order by id")
```

Using IN queries with dynamic values

Available as of Camel 2.17

From Camel 2.17 onwards the SQL producer allows to use SQL queries with IN statements where the IN values is dynamic computed. For example from the message body or a header etc.

To use IN you need to:

- prefix the parameter name with `in`:
- add `()` around the parameter

An example explains this better. The following query is used:

```
-- this is a comment select * from projects where project in (:in:names) order by id
In the following route:
from("direct:query") .to("sql:classpath:sql/selectProjectsIn.sql") .to("log:query") .to("mock:query");
Then the IN query can use a header with the key names with the dynamic values such as:
// use an array template.requestBodyAndHeader("direct:query", "Hi there!", "names", new String[]{"Camel", "AMQ"}); // use a list
List<String> names = new ArrayList<String>(); names.add("Camel"); names.add("AMQ");
template.requestBodyAndHeader("direct:query", "Hi there!", "names", names); // use a string separated values with comma
template.requestBodyAndHeader("direct:query", "Hi there!", "names", "Camel,AMQ");
The query can also be specified in the endpoint instead of being externalized (notice that externalizing makes maintaining the
SQL queries easier)
from("direct:query") .to("sql:select * from projects where project in (:in:names) order by id") .to("log:query") .to("mock:query");
```

USING THE JDBC BASED IDEMPOTENT REPOSITORY

Available as of Camel 2.7: In this section we will use the JDBC based idempotent repository.

Abstract class From Camel 2.9 onwards there is an abstract class `org.apache.camel.processor.idempotent.jdbc.AbstractJdbcMessageIdRepository` you can extend to build custom JDBC idempotent repository.

First we have to create the database table which will be used by the idempotent repository. For **Camel 2.7**, we use the following schema:

```
sqlCREATE TABLE CAMEL_MESSAGEPROCESSED ( processorName VARCHAR(255), messageId VARCHAR(100) )
```

In **Camel 2.8**, we added the `createdAt` column:

sqlCREATE TABLE CAMEL_MESSAGEPROCESSED (processorName VARCHAR(255), messageId VARCHAR(100), createdAt TIMESTAMP) The SQL Server **TIMESTAMP** type is a fixed-length binary-string type. It does not map to any of the JDBC time types: **DATE**, **TIME**, or **TIMESTAMP**.

We recommend to have a unique constraint on the columns processorName and messageId. Because the syntax for this constraint differs for database to database, we do not show it here.

Second we need to setup a `javax.sql.DataSource` in the spring XML file:

```
xml<jdbc:embedded-database id="dataSource" type="DERBY" />
```

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

```
xml<bean id="messageIdRepository" class="org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository">
<constructor-arg ref="dataSource" /> <constructor-arg value="myProcessorName" /> </bean>
```

Customize the JdbcMessageIdRepository

Starting with **Camel 2.9.1** you have a few options to tune the `org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository` for your needs:

Parameter	Default Value	Description
createTableIfNotExists	true	Defines whether or not Camel should try to create the table if it doesn't exist.
tableExistsString	SELECT 1 FROM CAMEL_MESSAGEPROCESSED WHERE 1 = 0	This query is used to figure out whether the table already exists or not. It must throw an exception to indicate the table doesn't exist.
createString	CREATE TABLE CAMEL_MESSAGEPROCESSED (processorName VARCHAR(255), messageId VARCHAR(100), createdAt TIMESTAMP)	The statement which is used to create the table.
queryString	SELECT COUNT(*) FROM CAMEL_MESSAGEPROCESSED WHERE processorName = ? AND messageId = ?	The query which is used to figure out whether the message already exists in the repository (the result is not equals to '0'). It takes two parameters. This first one is the processor name (<code>String</code>) and the second one is the message id (<code>String</code>).
insertString	INSERT INTO CAMEL_MESSAGEPROCESSED (processorName, messageId, createdAt) VALUES (?, ?, ?)	The statement which is used to add the entry into the table. It takes three parameter. The first one is the processor name (<code>String</code>), the second one is the message id (<code>String</code>) and the third one is the timestamp (<code>java.sql.Timestamp</code>) when this entry was added to the repository.
deleteString	DELETE FROM CAMEL_MESSAGEPROCESSED WHERE processorName = ? AND messageId = ?	The statement which is used to delete the entry from the database. It takes two parameter. This first one is the processor name (<code>String</code>) and the second one is the message id (<code>String</code>).

A customized `org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository` could look like:

```
xml<bean id="messageIdRepository" class="org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository">
<constructor-arg ref="dataSource" /> <constructor-arg value="myProcessorName" /> <property name="tableExistsString" value="SELECT 1 FROM CUSTOMIZED_MESSAGE_REPOSITORY WHERE 1 = 0" /> <property name="createString" value="CREATE TABLE CUSTOMIZED_MESSAGE_REPOSITORY (processorName VARCHAR(255), messageId VARCHAR(100), createdAt TIMESTAMP)" /> <property name="queryString" value="SELECT COUNT(*) FROM CUSTOMIZED_MESSAGE_REPOSITORY WHERE processorName = ? AND messageId = ?" /> <property name="insertString" value="INSERT INTO CUSTOMIZED_MESSAGE_REPOSITORY (processorName, messageId, createdAt) VALUES (?, ?, ?)" /> <property name="deleteString" value="DELETE FROM CUSTOMIZED_MESSAGE_REPOSITORY WHERE processorName = ? AND messageId = ?" /> </bean>
```

Using the JDBC based aggregation repository

Available as of Camel 2.6

Using `JdbcAggregationRepository` in Camel 2.6 In Camel 2.6, the `JdbcAggregationRepository` is provided in the `camel-jdbc-aggregator` component. From Camel 2.7 onwards, the `JdbcAggregationRepository` is provided in the `camel-sql` component. `JdbcAggregationRepository` is an `AggregationRepository` which on the fly persists the aggregated messages. This ensures that you will not lose messages, as the default aggregator will use an in memory only `AggregationRepository`.

The `JdbcAggregationRepository` allows together with Camel to provide persistent support for the Aggregator.

It has the following options:

Option	Type	Description
dataSource	DataSource	Mandatory: The <code>javax.sql.DataSource</code> to use.
repositoryName	String	Mandatory: The name of the repository.
transactionManager	TransactionManager	Mandatory: The <code>org.springframework.tran</code>

lobHandler	LobHandler	transactions for the database. The Transaction option to use a vendor specific LobHandler, for example org.springframework.jdbc.support.lob.
returnOldExchange	boolean	Whether the get operation should return the old Exchange if false to optimize as we do not need the old Exchange.
useRecovery	boolean	Whether or not recovery is enabled. This option is used by Camel Aggregator automatic recover failed aggregated Exchanges.
recoveryInterval	long	If recovery is enabled then a background task will recover and resubmit. By default this interval is 10 minutes.
maximumRedeliveries	int	Allows you to limit the maximum number of redelivery attempts. If the Exchange will be moved to the dead letter option is disabled. If this option is used then the maximumRedeliveries option is disabled.
deadLetteruri	String	An endpoint uri for a Dead Letter Channel where the Exchange is moved if the recovery option is used.
storeBodyAsText	boolean	Camel 2.11: Whether to store the message body as String or not. If false storing the body in binary form.
headersToStoreAsText	List<String>	Camel 2.11: Allows to store headers as String instead of binary format.
jdbcOptimisticLockingExceptionMapper	jdbcOptimisticLockingExceptionMapper	Camel 2.12: Allows to plugin a custom org.apache.camel.processor.aggregation.ExceptionMapper to map vendor specific error codes to an optimistic locking exception. This option requires optimisticLocking to be enabled.

Optimistic Locking

Optimistic locking is set to on by default. If two exchanges attempt to insert at the same time an exception will be thrown, caught, converted to an OptimisticLockingException, and rethrown.

What is preserved when persisting

JdbcAggregationRepository will only preserve any serializable compatible data types. If a data type is not such a type it's dropped and a WARN is logged. And it only persists the Message body and the Message headers. The Exchange properties are **not** persisted.

From Camel 2.11 onwards you can store the message body and select(ed) headers as String in separate columns.

Recovery

The JdbcAggregationRepository will by default recover any failed Exchange. It does this by having a background tasks that scans for failed Exchanges in the persistent store. You can use the checkInterval option to set how often this task runs. The recovery works as transactional which ensures that Camel will try to recover and redeliver the failed Exchange. Any Exchange which was found to be recovered will be restored from the persistent store and resubmitted and send out again.

The following headers are set when an Exchange is being recovered/redelivered:

Header	Type	Description
Exchange.REDELIVERED	Boolean	Is set to true to indicate the Exchange is being redelivered.
Exchange.REDELIVERY_COUNTER	Integer	The redelivery attempt, starting from 1.

Only when an Exchange has been successfully processed it will be marked as complete which happens when the confirm method is invoked on the AggregationRepository. This means if the same Exchange fails again it will be kept retried until it succeeds.

You can use option maximumRedeliveries to limit the maximum number of redelivery attempts for a given recovered Exchange. You must also set the deadLetteruri option so Camel knows where to send the Exchange when the maximumRedeliveries was hit.

You can see some examples in the unit tests of camel-sql, for example this test.

Database

To be operational, each aggregator uses two tables: the aggregation and completed one. By convention the completed has the same name as the aggregation one suffixed with "_COMPLETED". The name must be configured in the Spring bean with the RepositoryName property. In the following example aggregation will be used.

The table structure definition of both tables are identical: in both cases a String value is used as key (**id**) whereas a Blob contains the exchange serialized in byte array.

However one difference should be remembered: the **id** field does not have the same content depending on the table.

In the aggregation table **id** holds the correlation Id used by the component to aggregate the messages. In the completed table, **id** holds the id of the exchange stored in corresponding the blob field.

Here is the SQL query used to create the tables, just replace "aggregation" with your aggregator repository name.

```
sql CREATE TABLE aggregation ( id varchar(255) NOT NULL, exchange blob NOT NULL, constraint aggregation_pk PRIMARY KEY (id) );
CREATE TABLE aggregation_completed ( id varchar(255) NOT NULL, exchange blob NOT NULL, constraint
```

aggregation_completed_pk PRIMARY KEY (id));

Storing body and headers as text

Available as of Camel 2.11

You can configure the `JdbcAggregationRepository` to store message body and select(ed) headers as String in separate columns. For example to store the body, and the following two headers `companyName` and `accountName` use the following SQL:

```
sqlCREATE TABLE aggregationRepo3 ( id varchar(255) NOT NULL, exchange blob NOT NULL, body varchar(1000), companyName varchar(1000), accountName varchar(1000), constraint aggregationRepo3_pk PRIMARY KEY (id) ); CREATE TABLE aggregationRepo3_completed ( id varchar(255) NOT NULL, exchange blob NOT NULL, body varchar(1000), companyName varchar(1000), accountName varchar(1000), constraint aggregationRepo3_completed_pk PRIMARY KEY (id) );
```

And then configure the repository to enable this behavior as shown below:

```
xml <bean id="repo3" class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository"> <property name="repositoryName" value="aggregationRepo3"/> <property name="transactionManager" ref="txManager3"/> <property name="dataSource" ref="dataSource3"/> <!-- configure to store the message body and following headers as text in the repo --> <property name="storeBodyAsText" value="true"/> <property name="headersToStoreAsText"> <list> <value>companyName</value> <value>accountName</value> </list> </property> </bean>
```

Codec (Serialization)

Since they can contain any type of payload, Exchanges are not serializable by design. It is converted into a byte array to be stored in a database BLOB field. All those conversions are handled by the `JdbcCodec` class. One detail of the code requires your attention: the `ClassLoaderAwareObjectInputStream`.

The `ClassLoaderAwareObjectInputStream` has been reused from the Apache ActiveMQ project. It wraps an `ObjectInputStream` and use it with the `ContextClassLoader` rather than the `currentThread` one. The benefit is to be able to load classes exposed by other bundles. This allows the exchange body and headers to have custom types object references.

Transaction

A Spring `PlatformTransactionManager` is required to orchestrate transaction.

Service (Start/Stop)

The `start` method verify the connection of the database and the presence of the required tables. If anything is wrong it will fail during starting.

Aggregator configuration

Depending on the targeted environment, the aggregator might need some configuration. As you already know, each aggregator should have its own repository (with the corresponding pair of table created in the database) and a data source. If the default `LobHandler` is not adapted to your database system, it can be injected with the `LobHandler` property.

Here is the declaration for Oracle:

```
xml <bean id="lobHandler" class="org.springframework.jdbc.support.lob.OracleLobHandler"> <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/> </bean> <bean id="nativeJdbcExtractor" class="org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor"/> <bean id="repo" class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository"> <property name="transactionManager" ref="transactionManager"/> <property name="repositoryName" value="aggregation"/> <property name="dataSource" ref="dataSource"/> <!-- Only with Oracle, else use default --> <property name="lobHandler" ref="lobHandler"/> </bean>
```

Optimistic locking

From **Camel 2.12** onwards you can turn on `optimisticLocking` and use this JDBC based aggregation repository in a clustered environment where multiple Camel applications shared the same database for the aggregation repository. If there is a race condition there JDBC driver will throw a vendor specific exception which the `JdbcAggregationRepository` can react upon. To know which caused exceptions from the JDBC driver is regarded as an optimistic locking error we need a mapper to do this. Therefore there is a `org.apache.camel.processor.aggregate.jdbc.JdbcOptimisticLockingExceptionMapper` allows you to implement your custom logic if needed. There is a default implementation

`org.apache.camel.processor.aggregate.jdbc.DefaultJdbcOptimisticLockingExceptionMapper` which works as follows:

The following check is done:

If the caused exception is an `SQLException` then the `SQLState` is checked if starts with 23.

If the caused exception is a `DataIntegrityViolationException`

If the caused exception class name has "ConstraintViolation" in its name.

optional checking for FQN class name matches if any class names has been configured

You can in addition add FQN classnames, and if any of the caused exception (or any nested) equals any of the FQN class names, then its an optimistic locking error.

Here is an example, where we define 2 extra FQN class names from the JDBC vendor.

```
xml <bean id="repo" class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository"> <property name="transactionManager" ref="transactionManager"/> <property name="repositoryName" value="aggregation"/> <property name="dataSource" ref="dataSource"/> <property name="jdbcOptimisticLockingExceptionMapper" ref="myExceptionMapper"/> </bean> <!-- use the default mapper with extra FQN class names from our JDBC driver --> <bean id="myExceptionMapper" class="org.apache.camel.processor.aggregate.jdbc.DefaultJdbcOptimisticLockingExceptionMapper"> <property name="classNames"> <util:set> <value>com.foo.sql.MyViolationException</value> </util:set> </property> <value>com.foo.sql.MyOtherViolationException</value> </util:set> </bean>
```

Endpoint See Also

SQL Stored Procedure

JDBC

TELEGRAM COMPONENT

Available as of Camel 2.18

The **Telegram** component provides access to the Telegram Bot API. It allows a Camel-based application to send and receive messages by acting as a Bot, participating in direct conversations with normal users, private and public groups or channels.

A Telegram Bot must be created before using this component, following the instructions at the Telegram Bot developers home. When a new Bot is created, the BotFather provides an **authorization token** corresponding to the Bot. The authorization token is a mandatory parameter for the camel-telegram endpoint.

Note

In order to allow the Bot to receive all messages exchanged within a group or channel (not just the ones starting with a / character), ask the BotFather to **disable the privacy mode**, using the /setprivacy command.

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

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

URI Format

```
telegram:type/authorizationToken[?options]
```

Options

The Telegram component has no options. However, the Telegram component does support 24 endpoint options, which are listed below:

Name	Group	Default	Description
authorizationToken	common		Required. The authorization token for using the bot (as the BotFather) e.g. 5512311211:AAHx45322dubed3343211fgr3t-H.
type	common		Required. The endpoint type. Currently only the <code>bot</code> type is supported.
bridgeErrorHandler	consumer	false	Allow for bridging the consumer to the Camel routing Error Handler which means any exceptions occurred while the consumer is trying to pickup incoming messages or the like, will pass to the Camel routing error handler and result in an ErrorMessage. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions that will be logged at WARN/ERROR level and ignored.
limit	consumer	100	Limit on the number of updates that can be received in a single polling request.
sendEmptyMessageOnIdle	consumer	false	If set to true, the consumer will send an empty message (no body) instead of a null message.
timeOut	consumer	10000	Timeout in seconds for long polling. Can be smaller if you want to poll more frequently. Long polling produces shorter response times.
exceptionHandler	consumer (advanced)		The consumer use a custom ExceptionHandler. Note: If the option is disabled, then this options is not in use. By default the consumer will deal with exceptions that will be logged at WARN/ERROR level and ignored.
pollStrategy	consumer (advanced)		A pluggable org.apache.camel.polling.PollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange has been created and being routed in Camel.
chatId	producer		The identifier of the chat that will receive the produced messages. Chat Ids can be first obtained from incoming messages e.g. when a telegram user starts a conversation with a bot its client sends automatically a /start message containing the chat id. It is an optional parameter as the chat id can be set dynamically for each outgoing message using the <code>exchangePattern</code> .
exchangePattern	advanced		Sets the default exchange pattern when creating an exchange.
synchronous	advanced	false	Sets whether synchronous processing should be strictly used or Camel is allowed to use asynchronous processing (if supported).
backoffForErrorThreshold	scheduler		The number of subsequent error polls (failed due some error) that should happen before the backoffMultiplier should kick-in.
backoffIdleThreshold	scheduler		The number of subsequent idle polls that should happen before the backoffMultiplier should kick-in.
backoffMultiplier	scheduler		To let the scheduled polling consumer back-off if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.
delay	scheduler	500	Milliseconds before the next poll. You can also specify time values using units such as: <ul style="list-style-type: none">• 60s (60 seconds)• 5s30s (5 minutes and 30 seconds)• 1h (1 hour)
greedy	scheduler	false	If greedy is enabled then the Scheduler's poll() consumer will run immediately again if the previous run polled 1 or more messages.
initialDelay	scheduler	1000	Milliseconds before the first poll starts. You can also specify time values using units such as: <ul style="list-style-type: none">• 60s (60 seconds)• 5s30s (5 minutes and 30 seconds)• 1h (1 hour)
runLogLevel	scheduler	TRACE	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.
scheduledExecutorService	scheduler		Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.
scheduler	scheduler	none	To use a consumer specific scheduler, set this to a class name that implements org.apache.camel.component.telegram.TimedConsumer.
scheduleProperties	scheduler		Properties to be passed to the consumer specific scheduler.
startAt	scheduler		Whether the scheduler should be auto started.
timeUnit	scheduler	ms	Time unit for initialDelay and delay options.
useFixedDelay	scheduler	true	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in Jdk for details .

Message Headers

Name	Description
CamelTelegramChatId	This header is used by the producer endpoint in order to resolve the chat id that will receive the message. The recipient chat id can be placed (in order of priority) in message body, in the <code>CamelTelegramChatIdHeader</code> or in the endpoint configuration (<code>chatId</code> option). This header is also present in all incoming messages.
CamelTelegramMimeType	This header is used to identify the media type when the outgoing message is composed of pure binary data. Possible values are strings or enum values belonging to the <code>org.apache.camel.component.telegram.TelegramMimeType</code> enumeration.
CamelTelegramMediaCaption	This header is used to provide a caption or title for outgoing binary messages.

Usage

The Telegram component supports both consumer and producer endpoints. It can also be used in **reactive chat-bot mode** (to consume, then produce messages).

Producer Example

The following is a basic example of how to send a message to a Telegram chat through the Telegram Bot API.

In Java DSL:

```
from("direct:start")
.to("telegram:bots/123456789:AAE_dLq5C19xwGjw3yiC2NvEURZcejK21-Q987654321:AAE_dLq5C19xw0mg5yiC2NvSrkt3wj5Q1-L");
```

or in Spring XML:

```
<route>
  <from uri="direct:start"/>
  <to uri="telegram:bots/123456789:AAE_dLq5C19xwGjw3yiC2NvEURZcejK21-Q987654321:AAE_dLq5C19xw0mg5yiC2NvSrkt3wj5Q1-L"/>
</route>
```

The code 123456789:AAE_dLq5C19xwGjw3yiC2NvEURZcejK21-Q987654321:AAE_dLq5C19xw0mg5yiC2NvSrkt3wj5Q1-L is the **authorization token** corresponding to the Bot.

When using the producer endpoint without specifying the **chat id** option, the target chat will be identified using information contained in the body or headers of the message. The following message bodies are allowed for a producer endpoint (messages of type **OutgoingXXXMessage** belong to the package `org.apache.camel.component.telegram.model`)

Java Type	Description
<code>OutgoingTextMessage</code>	To send a text message to a chat.
<code>OutgoingImageMessage</code>	To send a photo (JPG, PNG) to a chat.
<code>OutgoingAudioMessage</code>	To send a mp3 audio to a chat.
<code>OutgoingVideoMessage</code>	To send a mp4 video to a chat.
<code>byte[]</code>	To send any media type supported. It requires the Camel <code>CamelTelegramd</code> header to be set to the appropriate media type.
<code>String</code>	To send a text message to a chat. It gets converted automatically into a <code>outgoingTextMessage</code> .

Consumer Example

The following is a basic example of how to receive all messages that telegram users are sending to the configured Bot.

In Java DSL:

```
from("telegram:bots/123456789:AAE_dLq5C19xwGjw3yiC2NvEURZcejK21-Q987654321:AAE_dLq5C19xw0mg5yiC2NvSrkt3wj5Q1-L")
.bean(ProcessorBean.class)
```

or in Spring XML:

```
<route>
  <from uri="telegram:bots/123456789:AAE_dLq5C19xwGjw3yiC2NvEURZcejK21-Q987654321:AAE_dLq5C19xw0mg5yiC2NvSrkt3wj5Q1-L"/>
  <bean ref="myBean"/>
</route>

<bean id="myBean" class="com.example.MyBean"/>
```

The MyBean is a simple bean that will receive the messages:

```
public class MyBean {
    public void process(String message) {
        // or Exchange, or org.apache.camel.component.telegram.model.IncomingMessage (or both)
        // do process
    }
}
```

Supported types for incoming messages are

Java Type	Description
<code>IncomingMessage</code>	The full object representation of an incoming message
<code>String</code>	The content of the message, for text messages only

Reactive Chat-Bot Example

The reactive chat-bot mode is a simple way of using the Camel component to build a simple chat bot that replies directly to chat messages received from the Telegram users.

The following is a basic configuration of the chat-bot in Java DSL

```
from("telegram:bots/123456789:AAE_dLq5C19xwGjw3yiC2NvEURZcejK21-Q987654321:AAE_dLq5C19xw0mg5yiC2NvSrkt3wj5Q1-L")
.bean(ChatBotLogic.class)
.to("telegram:bots/123456789:AAE_dLq5C19xwGjw3yiC2NvEURZcejK21-Q987654321:AAE_dLq5C19xw0mg5yiC2NvSrkt3wj5Q1-L");
```

or in Spring XML

```
<route>
  <from uri="telegram:bots/123456789:AAE_dLq5C19xwGjw3yiC2NvEURZcejK21-Q987654321:AAE_dLq5C19xw0mg5yiC2NvSrkt3wj5Q1-L"/>
```

```

<bean ref="chatBotLogic"/>
<to uri="telegram:bots/123456789:AAE_dLq5C19xwGjw3yiC2NvEURZcejK21-Q987654321:AAE_dLq5C19xw0mg5yiC2NvSrKT3wj5Q1-L"/>
</route>
<bean id="chatBotLogic" class="com.example.ChatBotLogic"/>
```

The ChatBotLogic is a simple bean that implements a generic String-to-String method.

```

public class ChatBotLogic {
    public String chatBotProcess(String message) {
        if( "do-not-reply".equals(message) ) {
            return null; // no response in the chat
        }
        return "echo from the bot: " + message; // echoes the message
    }
}
```

Every non-null string returned by the `chatBotProcess()` method is automatically routed to the chat that originated the request (as the `CamelTelegramChatId` header is used to route the message).

TEST COMPONENT

Testing of distributed and asynchronous processing is notoriously difficult. The Mock, Test and DataSet endpoints work great with the Camel Testing Framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's large range of Components together with the powerful Bean Integration.

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. That is, you use the test endpoint in a route and messages arriving on it will be implicitly compared to some expected messages extracted from some other location.

So you can use, for example, an expected set of message bodies as files. This will then set up a properly configured Mock endpoint, which is only valid if the received messages match the number of expected messages and their message payloads are equal.

Maven users will need to add the following dependency to their `pom.xml` for this component when using **Camel 2.8** or older:

```

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

From **Camel 2.9**: the Test component is provided directly in `camel-core`.

URI format

```
test:expectedMessagesEndpointUri
```

Where `expectedMessagesEndpointUri` refers to some other Component URI that the expected message bodies are pulled from before starting the test.

URI Options

Name	Default Value	Description
<code>anyorder</code>	false	Camel 2.17: Whether the expected messages should arrive in the same order, or in any order.
<code>delimiter</code>	\n\\v	Camel 2.17: The delimiter to use when <code>split=true</code> . The delimiter can be a regular expression.
<code>split</code>	false	Camel 2.17: If true messages loaded from the test endpoint will be split using the defined delimiter. For example to use a <code>f1 e</code> endpoint to load a file where each line is an expected message.
<code>timeout</code>	2000	Camel 2.12: The timeout to use when polling for message bodies from the URI.

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, your test case will perform the necessary assertions.

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 name is the name of the Timer object, which is created and shared across endpoints. So if you use the same name for all your timer endpoints, only one Timer object and thread will be used.

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

Note: The IN body of the generated exchange is null. So exchange.getIn().getBody() returns null.
Advanced Scheduler

See also the Quartz component that supports much more advanced scheduling.

Specify time in human friendly format

In **Camel 2.3** onwards you can specify the time in human friendly syntax.

Options

Name	Default Value	Description
time	null	A Java.util.Date the first event should be generated. If using the URI, the pattern expected is: www-yy-mm-dd HH:mm:ss or www-yy-mm-dd'T'HH:mm:ss.
pattern	null	Allows you to specify a custom date pattern to use for setting the time option using URI syntax.
period	1000	If greater than 0, generate periodic events every period milliseconds. The supported values are 100ms (100 milliseconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).
delay	0 / 1000	The number of milliseconds to wait before the first event is generated. Should not be used in conjunction with the time option. You can also specify time values in units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour). Before Camel 2.11 the default value is 0.
fixedDate	false	From Camel 2.11 the default value is false. From Camel 2.12 it is true. Allows you to specify a negative delay. In this scenario the timer will generate and fire events as soon as possible.
deadLetterUri	true	Specifies whether or not the thread associated with the timer should dies as a deadletter.
repeatCount	0	Camel 2.8: Specifies a maximum limit of number of fires. So if you set it to 1, this timer will only fire once. If you set it to 5, it will only fire five times. A value of zero or negative means fire forever.

Exchange Properties

When the timer is fired, it adds the following information as properties to the Exchange:

Name	Type	Description
Exchange.TIMER_NAME	String	The value of the name option.
Exchange.TIMER_TIME	Date	The value of the time option.
Exchange.TIMER_PERIOD	Long	The value of the period option.
Exchange.TIMER_FIXED_TIME	Date	The time when the consumer fired.
Exchange.TIMER_COUNTER	Long	Camel 2.8: The current fire counter. Starts from 0.

Message Headers

When the timer is fired, it adds the following information as headers to the IN message

Name	Type	Description
Exchange.TIMER_FIRED_TIME	java.util.date	The time when the consumer fired.

Sample

To set up a route that generates an event every 60 seconds:

```
from("timer://foo?fixedRate=true&period=60000").to("bean:myBean?method=someMethodName");
```

Instead of 60000 you can use period=60s which is more friendly to read.

The above route will generate an event and then invoke the someMethodName method on the bean called myBean in the Registry such as JNDI or Spring.

And the route in Spring DSL:

```
<route>
  <from uri="timer://foo?fixedRate=true&period=60000"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

Firing as soon as possible

Available as of Camel 2.17

You may want to fire messages in a Camel route as soon as possible you can use a negative delay:

```
<route>
  <from uri="timer://foo?delay=-1"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

In this way the timer will fire messages immediately.

You can also specify a repeatCount parameter in conjunction with a negative delay to stop firing messages after a fixed number has been reached.

If you don't specify a repeatCount then the timer will continue firing messages until the route will be stopped.

Firing only once

Available as of Camel 2.8

You may want to fire a message in a Camel route only once, such as when starting the route. To do that you use the repeatCount option as shown:

```
<route>
  <from uri="timer://foo?repeatCount=1"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

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 and based on any of the supported XML schema languages, which defaults to XML Schema

Note that the Jing component also supports the following useful schema languages:

- 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

Maven users will need to add the following dependency to their pom.xml for this component when using **Camel 2.8** or older:
`xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-spring</artifactId> <version>x.x.x</version> <!-- use the same version as your Camel core version --> </dependency>`

From Camel 2.9 onwards the Validation component is provided directly in the camel-core.

Options

confluenceTableSmall

Option	Default	Description
resourceResolverFactory	DefaultValidatorResourceResolverFactory	Camel 2.17: Reference to a org.apache.camel.component.validator.validatorResourceResolverFactory which creates a resource resolver per endpoint. The default creates an instance of

resourceResolver	null	org.apache.camel.component.validator.DefaultTLSR endpoint which creates the default resource resolver org.apache.camel.component.validator.DefaultTLSR default resource resolver reads the schema files from file system. This option instead of the option resourceRes the resource resolver depends on the resource URI of the specified in the endpoint; for example, if you want to extend resolver. This option is also available on the validator com set the resource resolver factory only once for all endpoint
useDom	false	Camel 2.9: Reference to a org.w3c.dom.ls.LSResource Whether DOMSource/DOMResult or SAXSource/SAXResult validator.
useSharedSchema	true	Camel 2.3: Whether the Schema instance should be shared introduced to work around a JDK 1.6.x bug. Xerces should
failOnNullBody	true	Camel 2.9.5/2.10.3: Whether to fail if no body exists.
headerName	null	Camel 2.11: To validate against a header instead of the message body.
failOnNullHeader	true	Camel 2.11: Whether to fail if no header exists when validating the message body.

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). {snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/component/validator/camelContext.xml}

Advanced: JMX method `clearCachedSchema`

Since **Camel 2.17**, you can force that the cached schema in the validator endpoint is cleared and reread with the next process call with the JMX operation `cLEARCACHEDSCHEMA`. You can also use this method to programmatically clear the cache. This method is available on the `ValidatorEndpoint` class.

Advanced: Global Option "CamelXmlValidatorAccessExternalDTD"

Since **Camel 2.19**, **2.18.3**, and **2.17.6** the default schema factory no longer allows reading external DTDs and external DTD entities. To achieve the old behavior where it was possible to access external DTDs and DTDs entities you can set the CamelContext global option "CamelXmlValidatorAccessExternalDTD" to "true". Prior to 2.19 global options were called properties.

Endpoint See Also

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.

Maven users will need to add the following dependency to their pom.xml for this component:
`<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-velocity</artifactId> <version>x.x.x</version> <!-- use the same version as your Camel core version --> </dependency>`

URI format

`velocity:templateName[?options]`

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (eg: `file://folder/myfile.vm`).

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

Options

confluenceTableSmall

Option	Default	Description
loaderCache	true	Velocity based file loader cache.
contentCache	true	Cache for the resource content when it is loaded. Note : as of Camel 2.9 cached resource content can be cleared via JMX using the endpoint's <code>cLEARCONTENTCACHE</code> operation.
encoding	null	Character encoding of the resource content.
propertiesFile	null	

New option in Camel 2.1: The URI of the properties file which is used for VelocityEngine initialization.

Message Headers

The velocity component sets a couple headers on the message (you can't set these yourself and from Camel 2.1 velocity component will not set these headers which will cause some side effect on the dynamic template support):
confluenceTableSmall

Header	Description
CamelVelocityResourceuri	The <code>templateName</code> as a <code>String</code> object.
CamelVelocitySupplementalcontext	Camel 2.16: To add additional information to the used VelocityContext. The value of this header should be a Map with key/values that will added (override any existing key with the same name). This can be used to pre setup some common key/values you want to reuse in your velocity endpoints.

Headers set during the Velocity evaluation are returned to the message and added as headers. Then its kinda possible to return values from Velocity to the Message.

For example, to set the header value of `fruit` in the Velocity template `.tm`:
`$in.setHeader("fruit", "Apple")`

The `fruit` header is now accessible from the `message.out.headers`.

Velocity Context

Camel will provide exchange information in the Velocity context (just a map). The Exchange is transferred as:
confluenceTableSmall

key	value
exchange	The Exchange itself.
exchange.properties	The Exchange properties.
headers	The headers of the In message.
camelContext	The Camel Context instance.
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).

Since Camel-2.14, you can setup a custom Velocity Context yourself by setting the message header `CamelVelocityContext` just like this

```
java VelocityContext velocityContext = new VelocityContext(variableMap); exchange.getIn().setHeader("CamelVelocityContext", velocityContext);
```

Hot reloading

The Velocity template resource is, by default, hot reloadable for both file and classpath resources (expanded jar). If you set `contentCache=true`, Camel will only load the resource once, and thus hot reloading is not possible. This scenario can be used in production, when the resource never changes.

Dynamic templates

Available as of Camel 2.1

Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.
confluenceTableSmall

Header	Type	Description
CamelVelocityResourceUri	String	Camel 2.1: A URI for the template resource to use instead of the endpoint configured.
CamelVelocityTemplate	String	Camel 2.1: The template to use instead of the endpoint configured.

Samples

For example you could use something like

```
from("activemq:My.Queue").to("velocity:com/acme/MyResponse.vm");
To use a Velocity template to formulate a response to a message for InOut message exchanges (where there is a JMSReplyTo header).
```

If you want to use InOnly and consume the message and send it to another destination, you could use the following route:
`from("activemq:My.Queue").to("velocity:com/acme/MyResponse.vm").to("activemq:Another.Queue");`
And to use the content cache, e.g. for use in production, where the .vm template never changes:
`from("activemq:My.Queue").to("velocity:com/acme/MyResponse.vm?contentCache=true").to("activemq:Another.Queue");`
And a file based resource:
`from("activemq:My.Queue").to("velocity:file:///myfolder/MyResponse.vm?contentCache=true").to("activemq:Another.Queue");`
In **Camel 2.1** it's possible to specify what template the component should use dynamically via a header, so for example:
`from("direct:in").setHeader("CamelVelocityResourceUri").constant("path/to/my/template.vm").to("velocity:dummy");`
In **Camel 2.1** it's possible to specify a template directly as a header the component should use dynamically via a header, so for example:
`from("direct:in").setHeader("CamelVelocityTemplate").constant("Hi this is a velocity template that can do templating ${body}").to("velocity:dummy");`

The Email Sample

In this sample we want to use Velocity templating for an order confirmation email. The email template is laid out in Velocity as:
`Dear ${headers.lastName}, ${headers.firstName} Thanks for the order of ${headers.item}. Regards Camel Riders Bookstore
${body}`
And the java code:
`{snippet:id=e1|lang=java|url=camel/trunk/components/camel-velocity/src/test/java/org/apache/camel/component/velocity/VelocityLetterTest.java}`
Endpoint See Also

VM COMPONENT

The **vm:** component provides asynchronous SEDA behavior, exchanging messages on a BlockingQueue and invoking consumers in a separate thread pool.

This component differs from the Seda component in that VM supports communication across CamelContext instances - so you can use this mechanism to communicate across web applications (provided that camel-core.jar is on the system/boot classpath).

VM is an extension to the Seda component.

URI format

```
vm:queueName[?options]
```

Where queueName can be any string to uniquely identify the endpoint within the JVM (or at least within the classloader that loaded camel-core.jar)

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

Before Camel 2.3 - Same URI must be used for both producer and consumer

An exactly identical VM endpoint URI **must** be used for both the producer and the consumer endpoint. Otherwise, Camel will create a second VM endpoint despite that the queueName portion of the URI is identical. For example:

```
from("direct:foo").to("vm:bar?concurrentConsumers=5");
from("vm:bar?concurrentConsumers=5").to("file://output");
```

Notice that we have to use the full URI, including options in both the producer and consumer.

In Camel 2.4 this has been fixed so that only the queue name must match. Using the queue name bar, we could rewrite the previous example as follows:

```
from("direct:foo").to("vm:bar");
from("vm:bar?concurrentConsumers=5").to("file://output");
```

Options

See the Seda component for options and other important usage details as the same rules apply to the Vm component.

Samples

In the route below we send exchanges across CamelContext instances to a VM queue named `order.email`:

```
from("direct:in").bean(MyOrderBean.class).to("vm:order.email");
```

And then we receive exchanges in some other Camel context (such as deployed in another .war application):

```
from("vm:order.email").bean(MyOrderEmailSender.class);
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Seda

XMPP COMPONENT

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

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

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

URI format

```
xmpp://[login@]hostname[:port][/participant][?Options]
```

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

The component supports both producer and consumer (you can get messages from XMPP or send messages to XMPP). Consumer mode supports rooms starting.

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

Options

Name	Description
room	If this option is specified, the component will connect to MUC (Multi User Chat). Usually, the domain name for MUC is different from the login domain. For example, if you are superman@jabber.org and want to join the krypton room, then the room URL is krypton@conference.jabber.org. Note the conference part. It is not a requirement to provide the full room JID. If the room parameter does not contain the @ symbol, the domain part will be discovered and added by Camel
user	User name (without server name). If not specified, anonymous login will be attempted.
password	Password.
resource	XMPP resource. The default is Camel.
createAccount	If true, an attempt to create an account will be made. Default is false.
participant	JID (Jabber ID) of person to receive messages. room parameter has precedence over participant.
nickname	Use nickname when joining room. If room is specified and nickname is not, user will be used for the nickname.
serviceName	The name of the service you are connecting to. For Google Talk, this would be gmail.com.
testConnectionOnStartup	Camel 2.11 Specifies whether to test the connection on startup. This is used to ensure that the XMPP client has a valid connection to the XMPP server when the route starts. Camel throws an exception on startup if a connection cannot be established. When this option is set to false, Camel will attempt to establish a "lazy" connection when needed by a producer, and will poll for a consumer connection until the connection is established. Default is true.
connectionPollDelay	Camel 2.11 The amount of time in seconds between polls to verify the health of the XMPP connection, or between attempts to establish an initial consumer connection. Camel will try to re-establish a connection if it has become inactive. Default is 10 seconds.
pubsub	Camel 2.15 Accept pubsub packets on input, default is false
doc	Camel 2.15 Set a doc header on the IN message containing a Document form of the incoming packet; default is true if presence or pubsub are true, otherwise false
connectionConfiguration	Camel 2.18: To use an existing connection configuration

Headers and setting Subject or Language

Camel sets the message IN headers as properties on the XMPP message. You can configure a `HeaderFilterStrategy` if you need custom filtering of headers.

The `Subject` and `Language` of the XMPP message are also set if they are provided as IN headers.

Examples

User superman to join room krypton at jabber server with password, secret:

```
xmpp://superman@jabber.org/?room=krypton@conference.jabber.org&password=secret
```

User superman to send messages to joker:

```
xmpp://superman@jabber.org/joker@jabber.org?password=secret
```

Routing example in Java:

```
from("timer://kickoff?period=10000").
setBody(constant("I will win!\n Your Superman.")).
to("xmpp://superman@jabber.org/joker@jabber.org?password=secret");
```

Consumer configuration, which writes all messages from joker into the queue, `evil.talk`:

```
from("xmpp://superman@jabber.org/joker@jabber.org?password=secret").
to("activemq:evil.talk");
```

Consumer configuration, which listens to room messages:

```
from("xmpp://superman@jabber.org/?password=secret&room=krypton@conference.jabber.org").
to("activemq:krypton.talk");
```

Room in short notation (no domain part):

```
from("xmpp://superman@jabber.org/?password=secret&room=krypton").
to("activemq:krypton.talk");
```

When connecting to the Google Chat service, you'll need to specify the `serviceName` as well as your credentials:

```
from("direct:start").
to("xmpp://talk.google.com:5222/touser@gmail.com?serviceName=gmail.com&user=fromuser&password=secret").
to("mock:result");
```

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 responses for requests.

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

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

URI format

```
xquery:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template.

For example you could use something like this:

```
from("activemq:My.Queue").
  to("xquery:com/acme/mytransform.xquery");
```

To use an XQuery template to formulate a response to a message for InOut message exchanges (where there is a **JMSReplyTo** header).

If you want to use InOnly, consume the message, and send it to another destination, you could use the following route:

```
from("activemq:My.Queue").
  to("xquery:com/acme/mytransform.xquery").
  to("activemq:Another.Queue");
```

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 responses for requests.

URI format

```
xslt:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template. Refer to the Spring Documentation for more detail of the URI syntax

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

Here are some example URIs

URI	Description
xslt:com/acme/mytransform.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.xls	refers to the remote http resource

Maven users will need to add the following dependency to their pom.xml for this component when using **Camel 2.8** or older:

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

From Camel 2.9 onwards the XSLT component is provided directly in the camel-core.

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 <code>org.apache.camel.converter.jaxp.XmlConverter</code> .
transformerFactory	null	Option to override default TransformerFactory. Will lookup for the transformerFactory in the Registry. The provided transformer factory must be of type <code>javax.xml.transform.TransformerFactory</code> .
transformerFactoryClass	null	Option to override default TransformerFactory. Will create a <code>TransformerFactoryClass</code> instance and set it to the converter.
uriResolverFactory	DefaultXsltUriResolverFactory	Camel 2.17: Reference to a <code>org.apache.camel.component.xslt.XsltUriResolverFactory</code> which creates an URI resolver per endpoint. The default implementation <code>org.apache.camel.component.xslt.DefaultXsltUriResolverFactory</code> which creates the default URI resolver for each endpoint. But you can also provide your own <code>org.apache.camel.component.xslt.XsltUriResolverFactory</code> implementation if you want to do customizations on the classpath and the file system. This option instead of the option <code>uriResolver</code> shall be used when the URI resolver depends on the resource URI of the root XSLT document specified in the endpoint; for example if you want to extend the default URI resolver. This option is also available on the XSLT component, so that you can set the resource resolver factory only once for all endpoints.
uriResolver	null	Camel 2.3: Allows you to use a custom <code>java.util.Transformation.UriResolver</code> . Camel will by default use its own implementation <code>org.apache.camel.bulider.util.XsltUriResolver</code> which is capable of reading from classpath.
resultHandlerFactory	null	Camel 2.3: Allows you to use a custom <code>org.apache.camel.builder.XsltResultHandlerFactory</code> which is capable of using custom <code>org.apache.camel.bulider.util.ResultHandler</code> types.
failOnNullBody	true	Camel 2.3: Whether or not to throw an exception if the input body is null.

<code>deleteOutputFile</code>	<code>false</code>	Camel 2.6: If you have <code>output->file</code> then this option dictates whether or not the output file should be deleted when the Exchange is done processing. For example suppose the output file is a temporary file, then it can be a good idea to delete it after use.
<code>output</code>	<code>string</code>	Camel 2.3: Option to specify which output type to use. Possible values are: <code>String</code> , <code>Bytes</code> , <code>DOM</code> , <code>File</code> . The first three options are all in memory based, whereas <code>File</code> is streamed directly to a <code>java.io.File</code> . For <code>File</code> you must specify the filename in the IN header with the key <code>Exchange.XSLT_FILE_NAME</code> which is also <code>CamelXsltFileName</code> .
<code>contentCache</code>	<code>true</code>	Camel 2.6: Cache for the resource content (the stylesheet file) when it is loaded. If set to <code>false</code> Camel will reload the stylesheet file on each message processing. This is useful for performance reasons. Note: from Camel 2.9 a cached stylesheet can be forced to reload at runtime via JMX using the <code>clearCachedStylesheet</code> operation.
<code>allowSAX</code>		Camel 2.3/2.9: Whether to allow using SAX as the <code>javax.xml.transform.Source</code> . The option is default <code>false</code> in Camel 2.11.3/2.12.2 or older. And default <code>true</code> in Camel 2.11.4/2.12.3 onwards.
<code>transformerCacheSize</code>	<code>0</code>	Camel 2.9.3/2.10.1: The number of <code>javax.xml.transform.Transformer</code> objects that are cached for reuse to avoid calls to <code>Template.newTransformer()</code> .
<code>saxon</code>	<code>false</code>	Camel 2.11: Whether to use Saxon as the <code>TransformerFactory</code> class. If enabled then the class <code>net.sf.saxon.TransformerFactoryImpl</code> . You would need to add Saxon to the classpath.
<code>saxonExtensionFunctions</code>	<code>null</code>	Camel 2.17: Allows to configure one or more custom <code>net.sf.saxon.lib.ExtensionFunctionDefinition</code> . You would need to add Saxon to the classpath. By setting this option, the extension functions will be available in the XSLT template.
<code>errorListener</code>		Camel 2.14: Allows to configure to use a custom <code>javax.xml.transform.ErrorListener</code> . Note: when doing this then the default error listener which captures any errors occurring during the transformation process is disabled. See also this option for specifying the error listener .
<code>entityResolver</code>		Camel 2.16: To use an external <code>EntityResolver</code> when performing transformations via SAXParser.

Using XSLT endpoints

For example you could use something like

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl");
```

To use an XSLT template to 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 the following route:

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl").
  to("activemq:Another.Queue");
```

Getting Parameters into the XSLT to work with

By default, all headers are added as parameters which are available in the XSLT. To do this you will need to declare the parameter so it is then *useable*.

```
<setHeader headerName="myParam"><constant>42</constant></setHeader>
<to uri="xslt:MyTransform.xsl"/>
```

And the XSLT just needs to declare it at the top level for it to be available:

```
<xsl: ..... >
<xsl:param name="myParam"/>
<xsl:template ...>
```

Spring XML versions

To use the above examples in Spring XML you would use something like

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="activemq:My.Queue"/>
    <to uri="xslt:org/apache/camel/spring/processor/example.xsl"/>
    <to uri="activemq:Another.Queue"/>
  </route>
</camelContext>
```

There is a test case along with its Spring XML if you want a concrete example.

Using `xsl:include`

Camel 2.2 or older

If you use `xsl:include` in your XSL files then in Camel 2.2 or older it uses the default `javax.xml.transform.URIResolver` which means it can only lookup files from file system, and its does that relative from the JVM starting folder.

For example this include:

```
<xsl:include href="staff_template.xsl"/>
```

Will lookup the `staff_template.xsl` file from the starting folder where the application was started.

Camel 2.3 or newer

Now Camel provides its own implementation of `URIResolver` which allows Camel to load included files from the classpath and more intelligent than before.

For example this include:

```
<xsl:include href="staff_template.xsl"/>
```

Will now be located relative from the starting endpoint, which for example could be:

```
.to("xslt:org/apache/camel/component/xslt/staff_include_relative.xsl")
```

Which means Camel will locate the file in the **classpath** as `org/apache/camel/component/xslt/staff_template.xsl`. This allows you to use xsl include and have xsl files located in the same folder such as we do in the example `org/apache/camel/component/xslt`.

You can use the following two prefixes `classpath:` or `file:` to instruct Camel to look either in classpath or file system. If you omit the prefix then Camel uses the prefix from the endpoint configuration. If that neither has one, then classpath is assumed.

You can also refer back in the paths such as

```
<xsl:include href="../staff_other_template.xsl"/>
```

Which then will resolve the xsl file under `org/apache/camel/component`.

Using xsl:include and default prefix

When using xsl:include such as:

```
<xsl:include href="staff_template.xsl"/>
```

Then in Camel 2.10.3 and older, then Camel will use "classpath:" as the default prefix, and load the resource from the classpath. This works for most cases, but if you configure the starting resource to load from file,

```
.to("xslt:file:etc/xslt/staff_include_relative.xsl")
```

.. then you would have to prefix all your includes with "file:" as well.

```
<xsl:include href="file:staff_template.xsl"/>
```

From Camel 2.10.4 onwards we have made this easier as Camel will use the prefix from the endpoint configuration as the default prefix. So from Camel 2.10.4 onwards you can do:

```
<xsl:include href="staff_template.xsl"/>
```

Which will load the `staff_template.xsl` resource from the file system, as the endpoint was configured with "file:" as prefix. You can still though explicitly configure a prefix, and then mix and match. And have both file and classpath loading. But that would be unusual, as most people either use file or classpath based resources.

Using Saxon extension functions

Since Saxon 9.2, writing extension functions has been supplemented by a new mechanism, referred to as integrated extension functions you can now easily use camel:

- Java example:

```
SimpleRegistry registry = new SimpleRegistry();
registry.put("function1", new MyExtensionFunction1());
registry.put("function2", new MyExtensionFunction2());

CamelContext context = new DefaultCamelContext(registry);
context.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
            .to("xslt:org/apache/camel/component/xslt/extensions/extensions.xslt?saxonExtensionFunctions=#function1,#function2");
    }
});
```

Spring example:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:extensions"/>
        <to uri="xslt:org/apache/camel/component/xslt/extensions/extensions.xslt?saxonExtensionFunctions=#function1,#function2"/>
    </route>
</camelContext>

<bean id="function1" class="org.apache.camel.component.xslt.extensions.MyExtensionFunction1"/>
<bean id="function2" class="org.apache.camel.component.xslt.extensions.MyExtensionFunction2"/>
```

Dynamic stylesheets

To provide a dynamic stylesheet at runtime you can define a dynamic URI. See How to use a dynamic URI in to() for more information.

Available as of Camel 2.9 (removed in 2.11.4, 2.12.3 and 2.13.0)

Camel provides the `CamelXsltResourceuri` header which you can use to define a stylesheet to use instead of what is configured on the endpoint URI. This allows you to provide a dynamic stylesheet at runtime.

Accessing warnings, errors and fatalErrors from XSLT ErrorListener

Available as of Camel 2.14

From Camel 2.14 onwards, any warning/error or fatalError is stored on the current Exchange as a property with the keys `Exchange.XSLT_ERROR`, `Exchange.XSLT_FATAL_ERROR`, or `Exchange.XSLT_WARNING` which allows end users to get hold of any errors happening during transformation.

For example in the stylesheet below, we want to terminate if a staff has an empty dob field. And to include a custom error message using `xsl:message`.

```
<xsl:template match="/">
<html>
<body>
<xsl:for-each select="staff/programmer">
<p>Name: <xsl:value-of select="name"/><br />
<xsl:if test="dob=&#39;&#39;">
<xsl:message terminate="yes">Error: DOB is an empty string!</xsl:message>
</xsl:if>
</p>
</xsl:for-each>
</body>
</html>
</xsl:template>
```

This information is not available on the Exchange stored as an Exception that contains the message in the `getMessage()` method on the exception. The exception is stored on the Exchange as a warning with the key `Exchange.XSLT_WARNING`.

Notes on using XSLT and Java Versions

Here are some observations from Sameer, a Camel user, which he kindly shared with us:

In case anybody faces issues with the XSLT endpoint please review these points.

I was trying to use an xslt endpoint for a simple transformation from one xml to another using a simple xsl. The output xml kept appearing (after the xslt processor in the route) with outermost xml tag with no content within.

No explanations show up in the DEBUG logs. On the TRACE logs however I did find some error/warning indicating that the XMLConverter bean could no be initialized.

After a few hours of cranking my mind, I had to do the following to get it to work (thanks to some posts on the users forum that gave some clue):

1. Use the `transformerFactory` option in the route ("`xslt:my-transformer.xsl?transformerFactory=tFactory`") with the `tFactory` bean having bean defined in the spring context for `class="org.apache.xalan.xsltc.trax.TransformerFactoryImpl"`.
2. Added the Xalan jar into my maven pom.

My guess is that the default xml parsing mechanism supplied within the JDK (I am using 1.6.0_03) does not work right in this context and does not throw up any error either. When I switched to Xalan this way it works. This is not a Camel issue, but might need a mention on the xslt component page.

Another note, jdk 1.6.0_03 ships with JAXB 2.0 while Camel needs 2.1. One workaround is to add the 2.1 jar to the `jre/lib/endorsed` directory for the jvm or as specified by the container.

Hope this post saves newbie Camel riders some time.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started