



USER GUIDE

Version 2.9.8

Copyright 2007-2013, Apache Software Foundation

Table of Contents

	Table of Contents.....	ii
Chapter I	Introduction	I
Chapter I	Quickstart.....	I
Chapter I	Getting Started.....	5
Chapter I	Architecture.....	14
Chapter I	Enterprise Integration Patterns.....	30
Chapter I	Cook Book	35
Chapter I	Tutorials.....	84
Chapter I	Language Appendix.....	138
Chapter I	DataFormat Appendix.....	185
Chapter I	Pattern Appendix.....	234
Chapter I	Component Appendix	330
	Index	0

Introduction

Apache Camel ^a 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 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.

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:

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`:

or explicitly - as we do here when we add the `JMS Component`:

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:

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

This will start all of the configured routing rules.

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

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 of 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("jms:queue:orders").to("jms:queue:orders").end();
```

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="beanRef" class="org.springframework.samples.petclinic.util.BeanRef" />
```



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:

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.

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>`

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.

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:

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 world 😊) 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:

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

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.

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.

```
-----
```

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.

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

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

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()`.

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`.

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

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"

The first line in the above example creates an object which is an instance of an anonymous subclass of `RouteBuilder` with the specified `configure()` method.

The `CamelContext.addRoutes(RouterBuilder builder)` method invokes `builder.setContext(this)` 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 `RouterBuilder` classes.

Continue Learning about Camel

Return to the main [Getting Started](#) page for additional introductory reference information.

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
- XPath
- Mvel
- OGNL
- Ref Language
- 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.

Current Supported URIs

Component / ArtifactId / URI	Description
AHC / camel-ahc [redacted]	To call external HTTP services using Async Http Client
AMQP / camel-amqp [redacted]	For Messaging with AMQP protocol
APNS / camel-apns [redacted]	For sending notifications to Apple iOS devices
Atom / camel-atom [redacted]	Working with Apache Abdera for atom integration, such as consuming an atom feed.
Avro / camel-avro [redacted]	Working with Apache Avro for data serialization.
AWS-CW / camel-aws [redacted]	For working with Amazon's CloudWatch (CW).
AWS-DDB / camel-aws [redacted]	For working with Amazon's DynamoDB (DDB).

**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.

AWS-SDB /

camel-aws

For working with Amazon's SimpleDB (SDB).

AWS-SES /

camel-aws

For working with Amazon's Simple Email Service (SES).

AWS-SNS /

camel-aws

For Messaging with Amazon's Simple Notification Service (SNS).

AWS-SQS /

camel-aws

For Messaging with Amazon's Simple Queue Service (SQS).

AWS-S3 / camel-

aws

For working with Amazon's Simple Storage Service (S3).

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).

Bean Validation /
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

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.

Cache / camel-
cache

The cache component facilitates creation of caching endpoints and processors using EHCache as the cache implementation.

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).

CMIS / camel-cmis []	Uses the Apache Chemistry client API to interface with CMIS supporting CMS
Cometd / camel-cometd []	Used to deliver messages using the jetty cometd implementation of the bayeux protocol
Context / camel-context []	Used to refer to endpoints within a separate CamelContext to provide a simple black box composition approach so that routes 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 []	To integrate with Apache CouchDB.
Crypto (Digital Signatures) / camel-crypto []	Used to sign and verify exchanges using the Signature Service of the Java Cryptographic Extension.
CXF / camel-cxf []	Working with Apache CXF for web services integration
CXF Bean / camel-cxf []	Process 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 []	Working with Apache CXF for REST services integration
DataFormat / camel-core []	for working with Data Formats as if it was a regular Component supporting Endpoints and URIs.
DataSet / camel-core []	For load & soak testing the DataSet provides a way to create huge numbers of messages for sending to Components or asserting that they are consumed correctly
Direct / camel-core []	Synchronous call to another endpoint from same CamelContext.

Direct-VM / camel-core []	Synchronous call to another endpoint in another CamelContext running in the same JVM.
DNS / camel-dns []	To lookup domain information and run DNS queries using DNSJava
Disruptor / camel-disruptor []	To provide the implementation of SEDA which is based on disruptor
EJB / camel-ejb []	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.
ElasticSearch / camel- elasticsearch []	For interfacing with an ElasticSearch server.
Spring Event / camel-spring []	Working with Spring ApplicationEvents
EventAdmin / camel- eventadmin []	Receiving OSGi EventAdmin events
Exec / camel- exec []	For executing system commands
Facebook / camel-facebook []	Providing access to all of the Facebook APIs accessible using Facebook4J
File / camel-core []	Sending messages to a file or polling a file or directory.
Flatpack / camel- flatpack []	Processing fixed width or delimited files or messages using the FlatPack library
FOP / camel-fop []	Renders the message into different output formats using Apache FOP

FreeMarker / camel- freemarker <div></div>	Generates a response using a FreeMarker template
FTP / camel-ftp <div></div>	Sending and receiving files over FTP.
FTPS / camel-ftp <div></div>	Sending and receiving files over FTP Secure (TLS and SSL).
GAuth / camel- gae <div></div>	Used by web applications to implement an OAuth consumer. See also Camel Components for Google App Engine.
GHttp / camel- gae <div></div>	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.
GLogin / camel- gae <div></div>	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 <div></div>	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.
GMail / camel- gae <div></div>	Supports sending of emails via the mail service of Google App Engine. See also Camel Components for Google App Engine.
Geocoder / camel-geocoder <div></div>	Supports looking up geocoders for an address, or reverse lookup geocoders from an address.
Google Guava EventBus / camel-guava- eventbus <div></div>	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 <div></div>	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 <div></div>	For reading/writing from/to an HBase store (Hadoop database)

HDFS / camel-hdfs []	For reading/writing from/to an HDFS filesystem
HL7 / camel-hl7 []	For working with the HL7 MLLP protocol and the HL7 model using the HAPI library
HTTP / camel-http []	For calling out to external HTTP servers using Apache HTTP Client 3.x
HTTP4 / camel-http4 []	For calling out to external HTTP servers using Apache HTTP Client 4.x
iBATIS / camel-ibatis []	Performs a query, poll, insert, update or delete in a relational database using Apache iBATIS
IMAP / camel-mail []	Receiving email using IMAP
IRC / camel-irc []	For IRC communication
JavaSpace / camel-jaespace []	Sending and receiving messages through JavaSpace
jclouds / camel-jclouds []	For interacting with cloud compute & blobstore service via jclouds
JCR / camel-jcr []	Storing a message in a JCR compliant repository like Apache Jackrabbit
JDBC / camel-jdbc []	For performing JDBC queries and operations
Jetty / camel-jetty []	For exposing services over HTTP
JMS / camel-jms []	Working with JMS providers

JMX / camel-jmx []	For working with JMX notification listeners
JPA / camel-jpa []	For using a database as a queue via the JPA specification for working with OpenJPA, Hibernate or TopLink
Jsch / camel-jsch []	Support for the scp protocol
JT/400 / camel-jt400 []	For integrating with data queues on an AS/400 (aka System i, IBM i, i5, ...) system
Kestrel / camel-kestrel []	For producing to or consuming from Kestrel queues
Krati / camel-krati []	For producing to or consuming to Krati datastores
Language / camel-core []	Executes Languages scripts
LDAP / camel-ldap []	Performing searches on LDAP servers (<scope> must be one of object onelevel subtree)
Log / camel-core []	Uses Jakarta Commons Logging to log the message exchange to some underlying logging system like log4j
Lucene / camel-lucene []	Uses Apache Lucene to perform Java-based indexing and full text based searches using advanced analysis/tokenization capabilities
MINA / camel-mina []	Working with Apache MINA 1.x
MINA2 / camel-mina2 []	Working with Apache MINA 2.x
Mock / camel-core []	For testing routes and mediation rules using mocks

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
MQTT / camel- mqtt []	Component for communicating with MQTT M2M message brokers
MSV / camel-msv []	Validates the payload of a message using the MSV Library
Mustache / camel-mustache []	Generates a response using a Mustache template
MVEL / camel- mvel []	Generates a response using an MVEL template
MyBatis / camel- mybatis []	Performs a query, poll, insert, update or delete in a relational database using MyBatis
Nagios / camel- nagios []	Sending passive checks to Nagios using JSendNSCA
Netty / camel- netty []	Working with TCP and UDP protocols using Java NIO based capabilities offered by the Netty project
Netty HTTP / camel-netty-http []	Netty HTTP server and client using the Netty project
Pax-Logging / camel-paxlogging []	Receiving Pax-Logging events in OSGi
POP / camel-mail []	Receiving email using POP3 and JavaMail
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.

Properties / camel-core []	The properties component facilitates using property placeholders directly in endpoint uri definitions.
Quartz / camel-quartz []	Provides a scheduled delivery of messages using the Quartz 1.x scheduler
Quartz2 / camel-quartz2[?options] []	Provides a scheduled delivery of messages using the Quartz 2.x scheduler
Quickfix / camel-quickfix []	Implementation of the QuickFix for Java engine which allow to send/receive FIX messages
RabbitMQ / camel-rabbitmq []	Component for integrating with RabbitMQ
Ref / camel-core []	Component for lookup of existing endpoints bound in the Registry.
Restlet / camel-restlet []	Component for consuming and producing Restful resources using Restlet
RMI / camel-rmi []	Working with RMI
RNC / camel-jing []	Validates the payload of a message using RelaxNG Compact Syntax
RNG / camel-jing []	Validates the payload of a message using RelaxNG
Routebox / camel-routebox []	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 []	Working with ROME for RSS integration, such as consuming an RSS feed.
Salesforce / camel-salesforce []	To integrate with Salesforce

SAP NetWeaver / camel-sap- netweaver <div></div>	To integrate with SAP NetWeaver Gateway
SEDA / camel- core <div></div>	Asynchronous call to another endpoint in the same Camel Context
SERVLET / camel-servlet <div></div>	For exposing services over HTTP through the servlet which is deployed into the Web container.
SFTP / camel-ftp <div></div>	Sending and receiving files over SFTP (FTP over SSH).
Sip / camel-sip <div></div>	Publish/Subscribe communication capability using the Telecom SIP protocol. RFC3903 - Session Initiation Protocol (SIP) Extension for Event
SJMS / camel- sjms <div></div>	A ground up implementation of a JMS client
SMTP / camel- mail <div></div>	Sending email using SMTP and JavaMail
SMPP / camel- smpp <div></div>	To send and receive SMS using Short Messaging Service Center using the JSMPP library
SNMP / camel- snmp <div></div>	Polling OID values and receiving traps using SNMP via SNMP4J library
Solr / camel-solr <div></div>	Uses the Solrj client API to interface with an Apache Lucene Solr server
SpringBatch / camel-spring- batch <div></div>	To bridge Camel and Spring Batch
SpringIntegration / camel-spring- integration <div></div>	The bridge component of Camel and Spring Integration

Spring LDAP / camel-spring-ldap []	Camel wrapper for Spring LDAP
Spring Redis / camel-spring-redis []	Component for consuming and producing from Redis key-value store
Spring Web Services / camel-spring-ws []	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 []	Performing SQL queries using JDBC
SSH component / camel-ssh []	For sending commands to a SSH server
StAX / camel-stax []	Process messages through a SAX ContentHandler.
Stream / camel-stream []	Read or write to an input/output/error/file stream rather like unix pipes
Stomp / camel-stomp []	For communicating with Stomp compliant message brokers, like Apache ActiveMQ or ActiveMQ Apollo
StringTemplate / camel-stringtemplate []	Generates a response using a String Template
Stub / camel-core []	Allows you to stub out some physical middleware endpoint for easier testing or debugging
TCP / camel-mina []	Working with TCP protocols using Apache MINA

Test / camel-spring []	Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint
Timer / camel-core []	A timer endpoint
Twitter / camel-twitter []	A twitter endpoint
UDP / camel-mina []	Working with UDP protocols using Apache MINA
Validation / camel-core (camel-spring for Camel 2.8 or older) []	Validates the payload of a message using XML Schema and JAXP Validation
Velocity / camel-velocity []	Generates a response using an Apache Velocity template
Vertx / camel-vertx []	Working with the vertx event bus
VM / camel-core []	Asynchronous call to another endpoint in the same JVM
Weather / camel-weather []	Polls the weather information from Open Weather Map
Websocket / camel-websocket []	Communicating with Websocket clients
XML Security / camel-xmlsecurity []	Used to sign and verify exchanges using the XML signature specification.

XMPP / camel-xmpp [redacted]	Working with XMPP and Jabber
XQuery / camel-saxon [redacted]	Generates a response using an XQuery template
XSLT / camel-core (camel-spring for Camel 2.8 or older) [redacted]	Generates a response using an XSLT template
Yammer / camel-yammer [redacted]	Allows you to interact with the Yammer enterprise social network
Zookeeper / camel-zookeeper [redacted]	Working with ZooKeeper cluster(s)

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 [redacted]	Apache	For JMS Messaging with Apache ActiveMQ
ActiveMQ Journal / activemq-core [redacted]	Apache	Uses ActiveMQ's fast disk journaling implementation to store message bodies in a rolling log file
Activiti / activiti-camel [redacted]	Apache	For working with Activiti, a light-weight workflow and Business Process Management (BPM) platform which supports BPMN 2

Db4o / camel-db4o in camel-extra	GPL	For using a db4o datastore as a queue via the db4o library
----------------------------------	-----	--

Esper / camel-esper in camel-extra	GPL	Working with the Esper Library for Event Stream Processing
------------------------------------	-----	--

Hibernate / camel-hibernate in camel-extra	GPL	For using a database as a queue via the Hibernate library
--	-----	---

JBi / servicemix-camel	Apache	For JBi integration such as working with Apache ServiceMix
------------------------	--------	--

JCIFS / camel-jcifs in camel-extra	LGPL	This component provides access to remote file systems over the CIFS/SMB networking protocol by using the JCIFS library.
------------------------------------	------	---

JGroups / camel-jgroups in camel-extra	LGPL	The <code>jgroups</code> component provides exchange of messages between Camel infrastructure and JGroups clusters.
--	------	---

NMR / servicemix-nmr	Apache	Integration with the Normalized Message Router BUS in ServiceMix 4.x
----------------------	--------	--

RCode / camel-rcode in camel-extra	LGPL	Uses Rserve to integrate Camel with the statistics environment R
------------------------------------	------	--

Scalate / scalate-camel	Apache	Uses the given Scalate template to transform the message
-------------------------	--------	--

Smooks / camel-smooks in camel-extra.	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 in camel-extra <div></div>	to be clarified	Component for producing to Neo4j datastore using the Spring Data Neo4j library
ZeroMQ / camel-zeromq in camel-extra. <div></div>	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

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

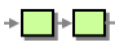
Messaging Systems



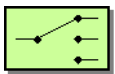
Message Channel How does one application communicate with another using messaging?



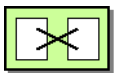
Message How can two applications connected by a message channel exchange a piece of information?



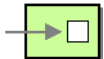
Pipes and Filters How can we perform complex processing on a message while maintaining independence and flexibility?



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?



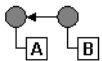



Message Endpoint How does an application connect to a messaging channel to send and receive messages?

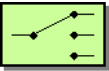
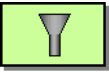
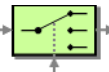
Messaging Channels

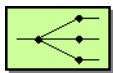
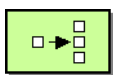
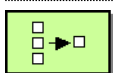
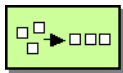
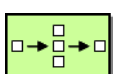
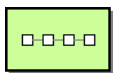
	Point to Point Channel	How can the caller be sure that exactly one receiver will receive the document or perform the call?
	Publish Subscribe Channel	How can the sender broadcast an event to all interested receivers?
	Dead Letter Channel	What will the messaging system do with a message it cannot deliver?
	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

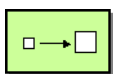
	Event Message	How can messaging be used to transmit events from one application to another?
	Request Reply	When an application sends a message, how can it get a response from the receiver?
	Correlation Identifier	How does a requestor that has received a reply know which request this is the reply for?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

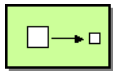
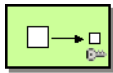
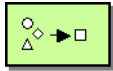
Message Routing

	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?

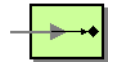



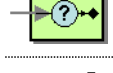



	Recipient List	How do we route a message to a list of (static or dynamically) specified recipients?
	Splitter	How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?
	Aggregator	How do we combine the results of individual, but related 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?
	Delayer	How can I delay the sending of a message?
	Load Balancer	How can I balance load across a number of endpoints?
	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?

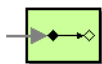
Message Transformation

	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?
	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

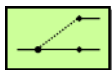
How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?

System Management



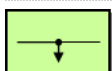
ControlBus

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

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.

- 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

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.

**Example**

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

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.

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



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.

Example:

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:

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

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

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 concatenates the fullname of the user into the fullName parameter.

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.

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`.

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.

We can use `Exchange` as well:

You can also have multiple types:

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

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:

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`:

@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.

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:

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:

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

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:

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:

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`:

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

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:

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 `asXml`. To invoke the `asXml` method we can do as follows:

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

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:

Listing 1. MyBean

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:

Listing 1. Invoke 2 parameter method

We can also use a `*` as wildcard so we can just say we want to execute the method with 2 parameters we do

Listing 1. Invoke 2 parameter method using wildcard

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 Injection

We support the injection of various resources using `@EndpointInject`. 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

Parameter Binding Annotations

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
<code>@Body</code>	To bind to an inbound message body	Ê
<code>@ExchangeException</code>	To bind to an Exception set on the exchange	Ê
<code>@Header</code>	To bind to an inbound message header	String name of the header
<code>@Headers</code>	To bind to the Map of the inbound message headers	Ê
<code>@OutHeaders</code>	To bind to the Map of the outbound message headers	Ê
<code>@Property</code>	To bind to a named property on the exchange	String name of the property
<code>@Properties</code>	To bind to the property map on the exchange	Ê
<code>@Handler</code>	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.	Ê



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

This may change in the future.



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

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.

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

The routing DSL then looks like this

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

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:

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:

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:

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

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

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 concatenates the fullname of the user into the `fullName` parameter.

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

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



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:

```
-----
```

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:

```
-----
```

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

```
-----
```

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

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.

For example

```
-----
```



This applies for them all

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

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:

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

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

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.

e.g. lets send a message to the **foo.bar** queue in ActiveMQ at some point

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

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. e.g.

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

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

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`,



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

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

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

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.

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

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

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`

In the above Foo interface the **`somelInOutMethod`** 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`

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

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

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 ec.

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.

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. 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 possible 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 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 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.

URI format

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&...



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.

Options

Option	Default	Description
<code>reportGroup</code>	<code>null</code>	A size to use a throughput logger for reporting

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.

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:

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:

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:

```

// ...

```

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:

```

// ...

```

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. Suppose you have the given route below:

Listing 1. Route

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:

Listing 1. adviceWith mocking all endpoints

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:

```

// ...

```

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

Listing 1. adviceWith mocking only log endpoints using a pattern

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.



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.



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.



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.

Listing 1. isMockEndpoints using camel-test kit

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:

Listing 1. 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:

Listing 1. test-camel-route.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:

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`".

Listing 1. adviceWith mock and skip sending to endpoints

The same example using the Test Kit

Listing 1. isMockEndpointsAndSkip using camel-test kit

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.

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.

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:

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

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

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

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [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.



time units

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

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 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.
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. Notice camel-test-spring is a new component in Camel 2.10 onwards. 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	Uses Guice to dependency inject your test classes
Camel TestNG	camel-testng	Supports plain TestNG based tests—with or without Spring or Guice—for Dependency Injection—which does not require an in-depth knowledge of Spring + Spring Test or Guice. Also from Camel 2.10 onwards, 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.

Notice how it derives from the Camel helper class **CamelTestSupport** but has no Spring or Guice dependency injection configuration but instead overrides the **createRouteBuilder()** method.

Spring Test with XML Config Example

Here is the Spring Testing example using XML Config.

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 FilterTest-context.xml on the classpath to configure the test case which looks like this

Spring Test with Java Config Example

Here is the Spring Testing example using Java Config.

For more information see Spring Java Config.

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.

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.

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.

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

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 5 messages. You can build complex expressions to match your criteria when to be notified.
AdviceWith	Allows you to advise or enhance an existing route using a RouteBuilder 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 Spring Testing or Guice the **camel-test** module was introduced so you can perform powerful Testing of your Enterprise Integration Patterns easily.

Adding to your pom.xml

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

JUnit

```

<code>
</code>

```



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

TestNG

Available as of Camel 2.8

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).

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.

Notice 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
boolean isUseRouteBuilder()	If the route builders from returned from createRouteBuilder() or createRouteBuilders() should be added to the CamelContext the test should be started.

boolean isUseAdviceWith()	<p>If the CamelContext use in the test should be automatic test methods are invoked.</p> <p>Override when using advice with and return true. This adviceWith is to be used, and the CamelContext will not be invoked before the advice with takes place. This delay helps by ensuring that the advice with has been property setup before the CamelContext is used.</p>
boolean isCreateCamelContextPerClass()	See Setup CamelContext once per class, or per every test class.
String isMockEndpoints()	<p>Triggers the auto-mocking of endpoints whose URIs match the default filter. The default filter is null which disables this feature. Returns the URIs of the endpoints.</p> <p>See org.apache.camel.impl.InterceptSendToMockEndpoint for details on the registration of the mock endpoints.</p>
boolean isUseDebugger()	<p>If this method returns true, the debugBefore(Exchange exchange, Processor processor, ProcessorDefinition<?> definition, String id, String label) and debugAfter(Exchange exchange, Processor processor, ProcessorDefinition<?> definition, String id, long timeTaken) methods are invoked for each processor in the routes.</p>
int getShutdownTimeout()	Returns the number of seconds that Camel should wait before shutdown. Useful for decreasing test times when a message is not received at the end of the test.
boolean useJmx()	If JMX should be disabled on the CamelContext used in the test.
JndiRegistry createRegistry()	Provides a hook for adding objects into the registry. Override to bind objects to the registry before test methods are invoked.
useOverridePropertiesWithPropertiesComponent	Camel 2.10: Allows to add/override properties when PropertyPlaceholder in Camel.
ignoreMissingLocationWithPropertiesComponent	Camel 2.10: Allows to control if Camel should ignore missing properties.

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 initialise the CamelContext.



It's important to start the CamelContext manually from the unit test after you are done doing all the advice with.

Here is an example `jndi.properties` file

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 runtime.

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. You may want to do this once, to share the CamelContext between test methods, to speedup unit testing. This requires to use 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:

Listing 1. Setup CamelContext once per class

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.



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 this fact. So use this feature with a bit of care.

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.xTestNG - Camel 2.8	Provided by org.apache.camel.test.CamelSpringTestSupport, org.apache.camel.testng.CamelSpringTestSupport. These support JUnit 3.x, JUnit 4.x and TestNG but do not support Spring annotations on the test classes.
Plain Spring Test	<ul style="list-style-type: none">JUnit 3.xJUnit 4.xTestNG	Extend the abstract base classes (org.springframework.test.context.junit38.AbstractJUnit38TestRunner, etc.) provided in Spring Test or use the Spring Test JUnit4 runner. This approach does not have feature parity with org.apache.camel.test.CamelSpringTestSupport.

Camel Enhanced Spring
Test

- JUnit 4.x -
**Camel
2.10**
- TestNG -
**Camel
2.10**

Use the `org.apache.camel.test.junit4.CamelSpringJUnit4C`
`org.apache.camel.testng.AbstractCamelTestNGSpringCo`
`org.apache.camel.test.junit4.CamelTestSupport` and also
and **@ContextConfiguration**.

CamelSpringTestSupport

`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.

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
but 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.

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 `FilterTest-context.xml` on the classpath to configure the test case which looks like this

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:

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.

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.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.

Camel Enhanced Spring Test

Using `org.apache.camel.test.junit4.CamelSpringJUnit4ClassRunner` runner with the `@RunWith` annotation or extending `org.apache.camel.testng.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 globally disabled during the test through the application contexts.
<code>org.apache.camel.test.spring.ExcludeRoutes</code>	Class	Indicates if certain route builder classes should be initialized. It initializes a <code>org.apache.camel.spi.PackageFilter</code> of given classes from being resolved. Type converters are excluded from the application contexts, which might otherwise be discovered and initialized.
<code>org.apache.camel.test.spring.LazyLoadTypeConverters</code> (Deprecated)	Class	Indicates if the CamelContexts that are loaded through the use of Spring Test loaded application contexts should use type converters.
<code>org.apache.camel.test.spring.MockEndpoints</code>	Class	Triggers the auto-mocking of endpoints via the <code>MockEndpoint</code> filter. The default filter is "*" which matches all endpoints. See <code>org.apache.camel.impl.InterceptSendToMockEndpoint</code> for details on the registration of the mock endpoints.
<code>org.apache.camel.test.spring.MockEndpointsAndSkip</code>	Class	Triggers the auto-mocking of endpoints via the <code>MockEndpoint</code> filter. The default filter is "*", which matches all endpoints. See <code>org.apache.camel.impl.InterceptSendToMockEndpoint</code> for details on the registration of the mock endpoints and skip sending the message to matched endpoints.
<code>org.apache.camel.test.spring.ProvidesBreakpoint</code>	Method	Indicates that the annotated method returns a <code>org.apache.camel.spi.Breakpoint</code> for use with the <code>MockEndpoint</code> for intercepting traffic to all endpoints or single endpoints for IDE for debugging. The method must be annotated with <code>ProvidesBreakpoint</code> and return <code>org.apache.camel.spi.Breakpoint</code> .
<code>org.apache.camel.test.spring.ShutdownTimeout</code>	Class	Indicates to set the shutdown timeout of the application contexts through the use of Spring Test loaded application contexts. If it is used, the timeout is automatically reduced to 10 seconds in the framework.
<code>org.apache.camel.test.spring.UseAdviceWith</code>	Class	Indicates the use of <code>adviceWith()</code> within the application contexts with this annotation and <code>UseAdviceWith</code> annotation. Any CamelContexts bootstrapped during the test loaded application contexts will not be started. The author is responsible for injecting the CamelContexts and executing <code>CamelContext#start()</code> on them. The advice has been applied to the routes in the application contexts.

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 that we still use the **@DirtyContext** annotation to ensure that the CamelContext, routes, and mock endpoints are reinitialized between test methods.

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.

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...

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.

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 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

Or inject them all

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 `JMSConnectionFactory` 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

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

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
- 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.

If you want to use InOnly and consume the message and send it to another destination you could use

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

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

```
-----
```

You can of course use the Xml Configuration if you prefer

ASYNCHRONOUS PROCESSING

Overview

Camel supports a more complex asynchronous processing model. The asynchronous processors implement the AsyncProcessor interface which is derived from the more synchronous 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 of 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.



Supported versions

The information on this page applies for Camel 2.4 onwards. Before Camel 2.4 the asynchronous processing is only implemented for JBI where as in Camel 2.4 onwards we have implemented it in many other areas. See more at [Asynchronous Routing Engine](#).

Interface Details

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:

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. See below for an example.

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, lets looks a what the thread model/sequence scenarios will look like for some sample routes.

The Jetty component's consumers support async processing by using continuations. Suffice to say it can take a http request and pass it to a camel route for async processing. If the processing is indeed async, it uses 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:

The sequence diagram would look something like this:

The diagram simplifies things by making it looks 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 2 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 async since it will use a NIO to complete getting 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.

Lets say we have 2 custom processors, `MyValidator` and `MyTransformation`, both of which are synchronous processors. Lets 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. Lets 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:

The sequence diagram would look something like this:

You would actually have multiple threads executing the 2nd 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 `AsyncProcessors` 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

- 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 use camel's routing engine and integration patterns support smoothly together with your CXF services.

INTEGRATE CAMEL INTO CXF TRANSPORT LAYER

To include the Camel Transport 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 application context 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.

```
<!-- Snippet for Spring application context -->
```

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.

```
<!-- Full example for programmatic integration -->
```

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/transport/camel`. It is commonly referred to using the prefix `camel`. 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.

Listing 1. Adding the Configuration Namespace

The destination element

You configure an Camel transport server endpoint using the `camel:destination` element and its children. The `camel:destination` element takes a single attribute, `name`, the specifies the WSDL port element that corresponds to the endpoint. The value for the `name` attribute takes the form `portQName.camel-destination`. The example below shows the `camel: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`.

Listing 1. camel:destination Element

The `camel:destination` element for Spring has a number of child elements that specify configuration information. They are described below.

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

The conduit element

You configure an 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`.

Listing 1. http-conf:conduit Element

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

Element	Description
---------	-------------

<code>camel-spring:camelContext</code>	You can specify the camel context in the camel conduit
<code>camel:camelContextRef</code>	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 namespace

`http://cxf.apache.org/transport/camel/blueprint` and import the schema like the blow.

Listing 1. Adding the Configuration Namespace for blueprint

In blueprint `camel:conduit` `camel:destination` only has one `camelContextId` attribute, they doesn't support to specify the camel context in the camel destination.

EXAMPLE USING CAMEL AS A LOAD BALANCER FOR CXF

This example show how to use the camel load balance feature in CXF, and you need load the configuration file in CXF and publish the endpoints on the address "camel://direct:EndpointA" and "camel://direct:EndpointB"

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:

```
-----
```

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.

- **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



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).

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

Ê

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.

- Tutorial on Spring Remoting with JMS
- Preface
- Prerequisites
- Distribution
- About
- Create the Camel Project
- Update the POM with Dependencies



Thanks

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

- Writing the Server
- Create the Spring Service
- Define the Camel Routes
- Configure Spring
- Run the Server
- Writing The Clients
- Client Using The ProducerTemplate
- Client Using Spring Remoting
- Client Using Message Endpoint EIP Pattern
- Run the Clients
- Using the Camel Maven Plugin
- Using Camel JMX
- See Also

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 the clients. This is archived with Camels powerful Endpoint pattern that even can be more powerful combined with Spring remoting. The tutorial have clients using each kind of technique for this.
Point to Point Channel	We using JMS queues so there are only one receive of the message exchange
Event Driven Consumer	Yes the JMS broker is of course event driven and only reacts 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

Update the POM with Dependencies

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

As we use spring xml configuration for the ActiveMQ JMS broker we need this dependency:



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.

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.

And the implementation of this service is:

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

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 scheme 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.

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:

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.

The ActiveMQ JMS broker is also configured in this xml file. We set it up to listen on TCP port 61610.

As this examples uses JMS then Camel needs a JMS component that is connected with the ActiveMQ broker. This is configured as shown below:

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 -PCamelServer
```

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.

```
.....
.....
.....
```

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 <code>ProducerTemplate</code> 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:

```
.....
```

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.

```
.....
```

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. **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:

```
.....
```

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!

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.

Listing 1. pom.xml

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:



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

LINKS

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

PART I

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:

```
[...]
```

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

```
[...]
```



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.

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`.

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

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

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:

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/`

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

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:

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.ReportIncidentEndpoint"` 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:

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 it's plugin

`org.mortbay.jetty:maven-jetty-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`.

Ê

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.

Ê

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.

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:

Ê

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

Ê

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.

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:

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:

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.

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:

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**.

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:

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

END OF PART I

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:

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

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 hearth of Camel its context.



Synchronize IDE

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

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

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.

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**:

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.

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:

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

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:

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



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.

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:

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

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:

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.

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`

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

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

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.

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.

Okay back to the 3 code lines:

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:

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**

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**:



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/camel/component/log` and its content is:

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.

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:

Letting Camel creating the mail body and storing it as a file is as easy as the following 3 code lines:

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:

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:

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 it's 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:

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:

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.

When we have the endpoint we can create the consumer (just as in part I 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.

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.

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:

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.

SENDING THE EMAIL

Sending the email requires access to a SMTP mail server, but the implementation code is very simple:

And just invoke the method from our consumer:

UNIT TESTING MAIL

For unit testing the consumer part we will use a mock mail framework, so we add this to our **pom.xml**:



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.



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.

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 repose
- the consumer that listen for mail files and send them as emails

So we change the constructor code a bit:

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:

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:

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:

Then we can use the template to create the file in **one code line**:

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:

The final class for the unit test is:

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.

ROUTING

Camel is particular 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 loosing control**. In my humble experience one of the key fears of developers is that they are forced into a tool/framework where they loose 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:

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 hearth 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



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! 😊

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:

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 hearth 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.

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:

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. 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

After adding it to the pom.xml you should refresh your Java Editor so it pickups the new jar. Then we are ready to create out unit test class.

We create this unit test skeleton, where we **extend** this class `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.

That is easy just return an instance of our route builder and this unit test will use our routes. 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.

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:

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 liked this. We could have used the **pipeline** verb to let out stand out that its the Pipes and Filters pattern such as:



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](#).



It is quite common in Camel itself to unit test using routes defined as an anonymous inner class, such as illustrated below:

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.

But most people are using the multi **.to** style instead.

We re-run our unit test and verifies that it still passes:

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 a 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\subfolder\ID-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 our webservice code to include the filename also:

However we could also have used the route builder itself to configure the constant filename as shown below:

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.

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. Phееeww 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:

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

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.

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.

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.



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:

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:

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:

And since we are on Java 1.5 we can use the static import of **ognl** so we have:

Notice the import static also applies for all the other languages, such as the Bean Language we used previously.

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.

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.

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.

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.

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.

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

- Tutorial using Axis 1.4 with Apache Camel
- Prerequisites
- Distribution
- Introduction
- Setting up the project to run Axis



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.

- 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:

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:

wsdl

We use the same .wsdl file as the Tutorial-Example-ReportIncident and copy it to `src/main/webapp/WEB-INF/wsdl`

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:

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:

The tool will generate the source code based on the wsdl and save the files to the following folder:

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.

Now we need to configure Axis itself and this is done using its `server-config.wsdd` file. We nearly get this for free from the auto generated code, we copy the stuff from `deploy.wsdd` and made a few modifications:

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**:

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:

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:

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**.

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:

Next is to add a plain spring XML file named **axis-example-context.xml** in the `src/main/resources` folder.

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.

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:

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:

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:

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:

Now that we have integrated with Spring then we easily integrate with Camel as Camel works well with Spring.

We choose to integrate Camel in the Spring XML file so we add the camel namespace and the schema location:

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:



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.

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:

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:

Then we can expose a `ProducerTemplate` property on our service with a setter in the Java code as:

And then let Spring handle the dependency inject as below:

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.

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 append /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

```
-----
```

And there should be a file in the target subfolder.

```
-----
```

Unit Testing

We would like to be able to unit test our **ReportIncidentService** class. So we add junit to the maven dependency:

```
-----
```

And then we create a plain junit testcase for our service class.

```
-----
```

Then we can run the test with maven using: `mvn test`. But we will get a failure:

```
-----
```

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:

```
-----
```

So now we are ready to set the ProducerTemplate on our service, and we get a hold of that baby from the CamelContext as:

```
-----
```

And this time when we run the unit test its a success:

```
-----
```

We would like to test that the file exists so we add these two lines to our test method:

```
-----
```

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:

And then we refactor our unit test to be a standard spring unit class. What we need to do is to extend `AbstractJUnit38SpringContextTests` 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:

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:

So we get hold of it by just getting it from the spring ApplicationContext as all spring users is used to do:

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:

Then we can create a new class **AxisReportIncidentServiceTest** to unit test with Jetty. The code to setup Jetty is shown below with code comments:

Now we just need to send the incident as a webservice request using Axis. So we add the following code:

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.

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:

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:

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:

To without the name:

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.

Step 1: Edit your web.xml

To enable spring add a context loader listener to your **/WEB-INF/web.xml** file

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

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...

You should update your Maven pom.xml to enable WAR packaging/naming like this...

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

Then you can run your web application as follows

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

TUTORIAL BUSINESS PARTNERS

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



Under Construction

This tutorial is a work in progress.

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

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:

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:

And a plugin configured:

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 I Input (XML over FTP)

To get a start on Customer I, we'll create an XSLT template to convert the Customer I 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 I sample input. You want to create an XSLT template to generate XML like the canonical XML sample above `<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:

```
<code><pre></pre></code>
```
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.
-
8. Put in an empty test method just for the moment (so when we run this we can see that "I test succeeded")
 9. Add the Spring `<beans>` element (including the Camel Namespace) with an empty `<camelContext>` element to the Spring context, like this:
-

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 I input file, and makes sure that some XML output comes out:

1. Save the input-customerI.xml file to `src/test/resources`
2. Save your XSLT file (created in the previous step) to `src/main/resources`
3. 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:
 1. Start from the endpoint `direct:start` (which lets the test conveniently pass messages into the route)
 2. Call the endpoint `xslt:YourXSLTFile.xsl` (to transform the message with the specified XSLT template)

3. Send the result to the endpoint `mock:finish` (which lets the test verify the route output)
4. Add a test method to the unit test class that:
 1. Get a reference to the Mock endpoint `mock:finish` using code like this:


```
MockEndpoint mockFinish = MockEndpoint.mock(mockFinishEndpoint);
```
 2. Set the `expectedMessageCount` on that endpoint to 1
 3. Get a reference to the Customer 1 input file, using code like this:


```
InputStream inputStream = getClass().getResourceAsStream("customer1.csv");
```
 4. Send that `InputStream` as a message to the `direct:start` endpoint, using code like this:


```
Context context = new Context();
context.setBody(inputStream);
MockEndpoint mockFinish.send(context);
```

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.

- 5. Ensure that the message made it through the route to the final endpoint, by testing all configured Mock endpoints like this:


```
MockEndpoint mockFinish.assertReceivedMessages(1);
```
- 6. 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.)
- 5. 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`

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



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.

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 `LineItemType` 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 `<td>` 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

Listing 1. CSVInputTest-context.xml

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:

```
<code><code>
```
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 in `Invoice` as before. The process should look something like this:

1. Update the Maven POM to include POI support:

```

<code>
</code>

```

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`
 - 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 `D` 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:
 - Remember to use a different start endpoint name if you're using the Java DSL and not use separate packages per test
 - You don't need the `unmarshal` step since the Excel POJO takes the raw `InputStream` from the source endpoint
 - 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.

Solution: Your finished test might look something like this:

- `src/test/java/org/apache/camel/tutorial/ExcelInputTest.java`
- For XML Configuration:
 - `src/test/resources/ExcelInputTest-context.xml`
- Or, for Java DSL Configuration:
 - `src/test/resources/ExcelInputTest-dsl-context.xml`
 - `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



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.

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.

So the idea is you specify a bean name which will then be resolved in the 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 attempted to be 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 from the Java DSL

```

// ...

```

Using Bean Expressions from XML

```

<!-- ...

```

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.

So we could implement it like this...

```

// ...

```

We can also use the Bean Integration annotations. For example you could do...

```

// ...

```

or

```

// ...

```

So you can bind parameters of the method to the Exchange, the Message or individual headers, properties, the body or other expressions.



Bean attribute now deprecated

Note, the `bean` attribute of the method expression element is now deprecated. You should now make use of `ref` attribute instead.

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. This is illustrated from the example below:

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 best suited method. If case of ambiguity Camel will thrown 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:

In Camel 2.2 onwards you can avoid the `BeanLanguage` and have it just as:

Which also can be done in a bit shorter and nice way:

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:

in this case, the Message coming from the `seda:a` Endpoint will have 'theHeader' header set to the constant value 'the value'.

And the same example using Java DSL:

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

You could also use slightly different syntax, e.g. if the header name is not a valid identifier:

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:

Dependencies

To use EL in your camel routes you need to add the a 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).

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:

In this case, the list of recipients are contained in the header 'myHeader'.

And the same example in Java DSL:

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)

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

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

Examples

Here is a simple example using a JXPath expression as a predicate in a Message Filter

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

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:

Dependencies

To use JXPath in your camel routes you need to add the a 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).

Otherwise, you'll also need Commons XPath.

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:

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

And the sample using Java DSL:

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:

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).

Otherwise, you'll also need MVEL

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:

Variables

Variable	Type	Description
this	Exchange	the Exchange is the root object
exchange	Exchange	the Exchange object
exception	Throwable	the Exchange exception (if any)
exchangelid	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

And the sample using Java DSL:

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:

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).

Otherwise, you'll also need OGNL

PROPERTY EXPRESSION LANGUAGE

The Property Expression Language allows you to extract values of named exchange properties.

Example usage

The recipientList element of the Spring DSL can utilize a property expression like:

In this case, the list of recipients are contained in the property 'myProperty'.

And the same example in Java DSL:

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)

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	el
Groovy	groovy
JavaScript	javaScript
JoSQL	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

The JSR-223 scripting languages ScriptContext is pre configured with the following attributes all set at `ENGINE_SCOPE`:

Attribute	Type	Value
context	<code>org.apache.camel.CamelContext</code>	The Camel Context
exchange	<code>org.apache.camel.Exchange</code>	The current Exchange

request	<code>org.apache.camel.Message</code>	The IN message
response	<code>org.apache.camel.Message</code>	The OUT message
properties	<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.

See Scripting Languages for the list of languages with explicit DSL support.

Additional arguments to ScriptingEngine

Available as of Camel 2.8

You can provide additional arguments to the `ScriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`. See this example:

```

// ...
// ...
// ...

```

Using properties function

Available as of 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, which key is provided in a header named `"foo"`.

```

// ...
// ...
// ...

```

From Camel 2.9 onwards you can now use the properties function and the same example is simpler:

```

// ...
// ...
// ...

```

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:

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).

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:

Or the something like this in your Spring XML:

You could follow the examples above to create an Predicate in a Message Filter or as an Expression for a Recipient List

ScriptContext

The JSR-223 scripting languages ScriptContext is pre configured with the following attributes all set at `ENGINE_SCOPE`:

Attribute	Type	Value
context	org.apache.camel.CamelContext	The Camel Context



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.

exchange	org.apache.camel.Exchange	The current Exchange
request	org.apache.camel.Message	The IN message
response	org.apache.camel.Message	The OUT message
		Camel 2.9: Function with a resolve method to make it easier to use Camels Properties component from scripts. See further below for example.
properties	org.apache.camel.builder.script.PropertiesFunction	

See Scripting Languages for the list of languages with explicit DSL support.

Additional arguments to ScriptingEngine

Available as of Camel 2.8

You can provide additional arguments to the `ScriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.

See this example:

Using properties function

Available as of 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, which key is provided in a header named "foo".

From Camel 2.9 onwards you can now use the properties function and the same example is simpler:

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:

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).

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

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.

And a Spring DSL sample as well:

ScriptContext

The JSR-223 scripting languages ScriptContext is pre configured with the following attributes all set at `ENGINE_SCOPE`:

Attribute	Type	Value
context	<code>org.apache.camel.CamelContext</code>	The Camel Context
exchange	<code>org.apache.camel.Exchange</code>	The current Exchange
request	<code>org.apache.camel.Message</code>	The IN message
response	<code>org.apache.camel.Message</code>	The OUT message
		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.
properties	<code>org.apache.camel.builder.script.PropertiesFunction</code>	

See Scripting Languages for the list of languages with explicit DSL support.

Additional arguments to ScriptingEngine

Available as of Camel 2.8

You can provide additional arguments to the `ScriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.
See this example:

Using properties function

Available as of 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, which key is provided in a header named `"foo"`.

From Camel 2.9 onwards you can now use the properties function and the same example is simpler:

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:

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).

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

For example you could use the **groovy** function to create an Predicate in a Message Filter or as an Expression for a Recipient List

Example

And the Spring DSL:

ScriptContext

The JSR-223 scripting languages ScriptContext is pre configured with the following attributes all set at `ENGINE_SCOPE`:

Attribute	Type	Value
context	<code>org.apache.camel.CamelContext</code>	The Camel Context
exchange	<code>org.apache.camel.Exchange</code>	The current Exchange
request	<code>org.apache.camel.Message</code>	The IN message
response	<code>org.apache.camel.Message</code>	The OUT message
properties	<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.

See Scripting Languages for the list of languages with explicit DSL support.

Additional arguments to ScriptingEngine

Available as of Camel 2.8

You can provide additional arguments to the `ScriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.

See this example:

Using properties function

Available as of 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, which key is provided in a header named `"foo"`.

From Camel 2.9 onwards you can now use the `properties` function and the same example is simpler:

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:

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).

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

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.

And a Spring DSL sample as well:

ScriptContext

The JSR-223 scripting languages ScriptContext is pre configured with the following attributes all set at `ENGINE_SCOPE`:

Attribute	Type	Value
context	<code>org.apache.camel.CamelContext</code>	The Camel Context
exchange	<code>org.apache.camel.Exchange</code>	The current Exchange
request	<code>org.apache.camel.Message</code>	The IN message
response	<code>org.apache.camel.Message</code>	The OUT message

`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.

See Scripting Languages for the list of languages with explicit DSL support.

Additional arguments to ScriptingEngine

Available as of Camel 2.8

You can provide additional arguments to the `ScriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.

See this example:

```
-----
```

Using properties function

Available as of 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, which key is provided in a header named `"foo"`.

```
-----
```

From Camel 2.9 onwards you can now use the `properties` function and the same example is simpler:

```
-----
```

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:

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).

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

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

The JSR-223 scripting languages ScriptContext is pre configured with the following attributes all set at `ENGINE_SCOPE`:

Attribute	Type	Value
context	org.apache.camel.CamelContext	The Camel Context
exchange	org.apache.camel.Exchange	The current Exchange
request	org.apache.camel.Message	The IN message
response	org.apache.camel.Message	The OUT message

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.

See Scripting Languages for the list of languages with explicit DSL support.

Additional arguments to ScriptingEngine

Available as of Camel 2.8

You can provide additional arguments to the `ScriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.

See this example:

```
-----
```

Using properties function

Available as of 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, which key is provided in a header named `"foo"`.

```
-----
```

From Camel 2.9 onwards you can now use the properties function and the same example is simpler:

```
-----
```

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:

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).

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

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.

And a Spring DSL sample as well:

ScriptContext

The JSR-223 scripting languages ScriptContext is pre configured with the following attributes all set at `ENGINE_SCOPE`:

Attribute	Type	Value
context	org.apache.camel.CamelContext	The Camel Context
exchange	org.apache.camel.Exchange	The current Exchange

request	<code>org.apache.camel.Message</code>	The IN message
response	<code>org.apache.camel.Message</code>	The OUT message
properties	<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.

See Scripting Languages for the list of languages with explicit DSL support.

Additional arguments to ScriptingEngine

Available as of Camel 2.8

You can provide additional arguments to the `ScriptingEngine` using a header on the Camel message with the key `CamelScriptArguments`.

See this example:

```

// Example code snippet

```

Using properties function

Available as of 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, which key is provided in a header named `"foo"`.

```

// Example code snippet

```

From Camel 2.9 onwards you can now use the properties function and the same example is simpler:

```

// Example code snippet

```

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:

```
-----
```

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).

```
-----
```

SIMPLE EXPRESSION LANGUAGE

The Simple Expression Language was a really simple language you can use, but has since grown more powerful. Its primarily intended for being a really small and simple language for evaluating Expression and Predicate without requiring any new dependencies or knowledge of XPath; so its ideal for testing in camel-core. Its ideal 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 `${body}` placeholders for complex expressions where the expression contains constant literals. The `${ }` placeholders can be omitted if the expression is only the token itself.

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 (i.e. having another `${ }`



Alternative syntax

From Camel 2.5 onwards 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 onwards 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 onwards, 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 onwards 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 is 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: `"${body} == 'Camel' "`. Notice how we have single quotes around the literal. The old style of using `"body"` and `"header.foo"` to refer to the message body and header is @deprecated, and its encouraged to always use `${ }` 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' "`.

placeholder in an existing, is not allowed).

From **Camel 2.9** onwards you can nest functions.

Variables

Variable	Type	Description
camelId	String	Camel 2.10: the CamelContext name
camelContext. OGNL	Object	Camel 2.11: the CamelContext invoked using a Camel OGNL expression.
exchangeId	String	Camel 2.3: the exchange id
id	String	the input message id
body	Object	the input body
in.body	Object	the input body
body. OGNL	Object	Camel 2.3: the input body invoked using a Camel OGNL expression.
in.body. OGNL	Object	Camel 2.3: the input body invoked using a Camel OGNL expression.
bodyAs(type)	Type	Camel 2.3: Converts the body to the given type determined by its classname. The converted body can be null.
mandatoryBodyAs(type)	Type	Camel 2.5: Converts the body to the given type determined by its classname, and expects the body to be not null.
out.body	Object	the output body
header.foo	Object	refer to the input foo header
header[foo]	Object	Camel 2.9.2: refer to the input foo header
headers.foo	Object	refer to the input foo header
headers[foo]	Object	Camel 2.9.2: refer to the input foo header
in.header.foo	Object	refer to the input foo header
in.header[foo]	Object	Camel 2.9.2: refer to the input foo header
in.headers.foo	Object	refer to the input foo header
in.headers[foo]	Object	Camel 2.9.2: refer to the input foo header
header.foo[bar]	Object	Camel 2.3: regard input foo header as a map and perform lookup on the map with bar as key
in.header.foo[bar]	Object	Camel 2.3: regard input foo header as a map and perform lookup on the map with bar as key
in.headers.foo[bar]	Object	Camel 2.3: regard input foo header as a map and perform lookup on the map with bar as key
header.foo. OGNL	Object	Camel 2.3: refer to the input foo header and invoke its value using a Camel OGNL expression.
in.header.foo. OGNL	Object	Camel 2.3: refer to the input foo header and invoke its value using a Camel OGNL expression.
in.headers.foo. OGNL	Object	Camel 2.3: refer to the input foo header and invoke its value using a Camel OGNL expression.
out.header.foo	Object	refer to the out header foo
out.header[foo]	Object	Camel 2.9.2: refer to the out header foo
out.headers.foo	Object	refer to the out header foo
out.headers[foo]	Object	Camel 2.9.2: refer to the out header foo
headerAs(key,type)	Type	Camel 2.5: Converts the header to the given type determined by its classname
headers	Map	Camel 2.9: refer to the input headers
in.headers	Map	Camel 2.9: refer to the input headers
property.foo	Object	refer to the foo property on the exchange
property[foo]	Object	Camel 2.9.2: refer to the foo property on the exchange
property.foo. OGNL	Object	Camel 2.8: refer to the foo property on the exchange and invoke its value using a Camel OGNL expression.
sys.foo	String	refer to the system property
sysenv.foo	String	Camel 2.3: refer to the system environment
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 (<code>Exchange.EXCEPTION_CAUGHT</code>) 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 (<code>Exchange.EXCEPTION_CAUGHT</code>) if the Exchange has any.
exception.stacktrace	String	Camel 2.6: Refer to the exception.stacktrace on the exchange, is null if no exception set on exchange. Will fallback and grab caught exceptions (<code>Exchange.EXCEPTION_CAUGHT</code>) if the Exchange has any.
date:command:pattern	String	Date formatting using the <code>java.text.SimpleDateFormat</code> patterns. Supported commands are: 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 <code>?method=methodname</code> syntax that is used by the Bean component.
properties:locations:key	String	Camel 2.3: Lookup a property with the given key. The <code>locations</code> option is optional. See more at Using PropertyPlaceholder.

routeId	String	Camel 2.1.1: 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 2.1.1: To refer to a type or field by its FQN name. To refer to a field you can append <code>.FIELD_NAME</code> . For example you can refer to the constant field from Exchange as: <code>org.apache.camel.Exchange.FILE_NAME</code>

OGNL expression support

Available as of Camel 2.3

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:

Camel understands the shorthand names for getters, but you can invoke any method or use the real name such as:

You can also use the null safe operator (`? .`) to avoid NPE if for example the body does NOT have an address

Its also possible to index in `Map` or `List` types, so you can do:

To assume the body is `Map` based and lookup the value with `foo` as key, and invoke the `getName` method on that value.

You can access the `Map` or `List` objects directly using their key name (with or without dots) :

Suppose there was no value with the key `foo` then you can use the null safe operator to avoid the NPE as shown:

You can also access `List` types, for example to get lines from the address you can do:

There is a special `last` keyword which can be used to get the last value from a list.

And to get the 2nd last you can subtract a number, so we can use `last-1` to indicate this:

And the 3rd last is of course:

And you can call the size method on the list with

From **Camel 2.11.1** onwards we added support for the length field for Java arrays as well, eg:

And yes you can combine this with the operator support as shown below:



Camel's OGNL support is for invoking methods only. You cannot access fields. From **Camel 2.11.1** onwards we added special support for accessing the length field of Java arrays.



If the key has space, then you **must** enclose the key with quotes, for example 'foo bar':

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:

Where the `rightValue` can be a String literal enclosed in `' '`, `null`, a constant value or another expression enclosed in `${ }`.

Camel will automatically type convert the `rightValue` type to the `leftValue` type, so its able to eg. convert a string into a numeric so you can use `>` comparison for numeric values.

The following operators are supported:

Operator	Description
<code>==</code>	equals
<code>></code>	greater than
<code>>=</code>	greater than or equals
<code><</code>	less than
<code><=</code>	less than or equals
<code>!=</code>	not equals
<code>contains</code>	For testing if contains in a string based value
<code>not contains</code>	For testing if not contains in a string based value
<code>regex</code>	For matching against a given regular expression pattern defined as a String value
<code>not regex</code>	For not matching against a given regular expression pattern defined as a String value
<code>in</code>	For matching if in a set of values, each element must be separated by comma.



Important

There **must** be spaces around the operator.

not in	For matching if not in a set of values, each element must be separated by comma.
is	For matching if the left hand side type is an instance of the value.
not is	For matching if the left hand side type is not an instance of the value.
range	For matching if the left hand side is within a range of values defined as numbers: <code>from..to</code> . From Camel 2.9 onwards the range values must be enclosed in single quotes.
not range	For matching if the left hand side is not within a range of values defined as numbers: <code>from..to</code> . From Camel 2.9 onwards the range values must be enclosed in single quotes.

And the following unary operators can be used:

Operator	Description
++	Camel 2.9: To increment a number by one. The left hand side must be a function, otherwise parsed as literal.
--	Camel 2.9: To decrement a number by one. The left hand side must be a function, otherwise parsed as literal.
\	Camel 2.9.3 to 2.10.x To escape a value, eg <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. Notice: Escaping is not supported using the File Language. Notice: From Camel 2.11 onwards the escape character is no longer support, but replaced with the following three special escaping.
<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.

And the following logical operators can be used to group expressions:

Operator	Description
and	deprecated use <code>&&</code> instead. The logical and operator is used to group two expressions.
or	deprecated use <code> </code> instead. The logical or operator is used to group two expressions.

&& **Camel 2.9:** The logical and operator is used to group two expressions.

|| **Camel 2.9:** The logical or operator is used to group two expressions.

The syntax for AND is:

And the syntax for OR is:

Some examples:

And a bit more advanced example where the right value is another expression

And an example with contains, testing if the title contains the word Camel

And an example with regex, testing if the number header is a 4 digit value:

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.

And for all the last 3 we also support the negate test using not:

And you can test if the type is a certain instance, eg for instance a String

We have added a shorthand for all `java.lang` types so you can write it as:

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:

Notice we use `..` in the range without spaces. Its based on the same syntax as Groovy.

From **Camel 2.9** onwards the range value must be in single quotes

Using and / or

If you have two expressions you can combine them with the `and` or `or` operator.

For instance:

And of course the `or` is also supported. The sample would be:



Using and,or operators

In **Camel 2.4 or older** the `and` or `or` can only be used **once** in a simple language expression. From **Camel 2.5** onwards you can use these operators multiple times.



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 1st 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:

```
int bar = 123;
String foo = "123";
foo < bar
```

which then ensures the `int` type is used as 1st 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.



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:

```
<simple>
  <when>
    <simple>
      <header>order.widget</header>
    </simple>
  </when>
</simple>
```



Camel 2.9 onwards

Use `&&` or `||` from Camel 2.9 onwards.

Notice: Currently `and` or `or` can only be used **once** in a simple language expression. This might change in the future.

So you **cannot** do:

```
<simple>
  <when>
    <simple>
      <header>order.widget</header>
    </simple>
  </when>
</simple>
```

Samples

In the Spring XML sample below we filter based on a header value:

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:fooOrders` endpoint, otherwise its lost in the deep blue sea 😊.

The same example in Java DSL:

You can also use the simple language for simple text concatenations such as:

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.

And in the sample below we invoke the bean language to invoke a method on a bean to be included in the returned string:

Where `orderIdGenerator` is the id of the bean registered in the Registry. If using Spring then its 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.

We can use the `?method=methodname` option that we are familiar with the Bean component itself:

And from Camel 2.3 onwards you can also convert the body to a given type, for example to ensure its a String you can do:

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`.

Its also possible to lookup a value from a header Map in **Camel 2.3** onwards:

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 onwards you can nest functions, such as shown below:

Referring to constants or enums

Available as of Camel 2.11

Suppose you have an enum for customers

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.

Using new lines or tabs in XML DSLs

Available as of Camel 2.9.3

From Camel 2.9.3 onwards its easier to specify new lines or tabs in XML DSLs as you can escape the value now

Setting result type

Available as of 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 useable to define types such as booleans, integers, etc.

For example to set a header as a boolean type you can do:

And in XML DSL

Changing function start and end tokens

Available as of 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:

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 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:

```
file:classpath:location
```

Dependencies

The Simple language is part of **camel-core**.

FILE EXPRESSION 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.

As opposed to Simple language File Language also supports Constant expressions so you can enter a fixed filename.

All the file tokens use the same expression name as the method on the `java.io.File` object, for instance `file:absolute` refers to the `java.io.File.getAbsolutePath()` method. Notice that not all expressions are supported by the current Exchange. For instance the FTP component supports some of the options, where as the File component supports all of them.

Expression	Type	File Consumer	File Producer	FTP Consumer	FTP Producer	Description
filename	String	yes	no	yes	no	refers to the file name (is relative to the starting directory, see note below)
filename.ext	String	yes	no	yes	no	Camel 2.3: refers to the file extension only
filename.noext	String	yes	no	yes	no	refers to the file name with no extension (is relative to the starting directory, see note below)
file:onlyname	String	yes	no	yes	no	refers to the file name only with no leading paths.
file:onlyname.noext	String	yes	no	yes	no	refers to the file name only with no extension and with no leading paths.
file:ext	String	yes	no	yes	no	refers to the file extension only
file:parent	String	yes	no	yes	no	refers to the file parent
file:path	String	yes	no	yes	no	refers to the file path



File language is now merged with Simple language

From Camel 2.2 onwards, the file language is now merged with Simple language which means you can use all the file syntax directly within the simple language.

file:absolute	Boolean	yes	no	no	no	refers to whether the file is regarded as absolute or relative
file:absolute.path	String	yes	no	no	no	refers to the absolute file path
file:length	Long	yes	no	yes	no	refers to the file length returned as a Long type
file:size	Long	yes	no	yes	no	Camel 2.5: refers to the file length returned as a Long type
file:modified	Date	yes	no	yes	no	refers to the file last modified returned as a Date type
date:command:pattern	String	yes	yes	yes	yes	for date formatting using the <code>java.text.SimpleDateFormat</code> patterns. Is an extension to the Simple language. Additional command is: file (consumers only) for the last modified timestamp of the file. Notice: all the commands from the Simple language can also be used.

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 will return as:

Expression	Returns
file:name	test\hello.txt
file:name.ext	txt
file:name.noext	test\hello
file:onlyname	hello.txt
file:onlyname.noext	hello
file:ext	txt
file:parent	filelanguage\test
file:path	filelanguage\test\hello.txt
file:absolute	false
file:absolute.path	\workspace\camel\camel-core\target\filelanguage\test\hello.txt

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 out endpoint to use the absolute starting directory `\workspace\camel\camel-core\target\filelanguage`. The file tokens will return as:

Expression	Returns
file:name	test\hello.txt
file:name.ext	txt
file:name.noext	test\hello
file:onlyname	hello.txt
file:onlyname.noext	hello
file:ext	txt
file:parent	\workspace\camel\camel-core\target\filelanguage\test
file:path	\workspace\camel\camel-core\target\filelanguage\test\hello.txt
file:absolute	true
file:absolute.path	\workspace\camel\camel-core\target\filelanguage\test\hello.txt

Samples

You can enter a fixed Constant expression such as `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 archieved using an expression like:

relative folder names are also supported so suppose the backup folder should be a sibling folder then you can append .. as:

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:

If you have a custom Date you want to use in the expression then Camel supports retrieving dates from the message header.

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:

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 PropertyPlaceholderConfigurer together 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:

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 onwards) to avoid clashing with Springs `PropertyPlaceholderConfigurer`.

Listing 1. bundle-context.xml

Listing 1. bundle-context.cfg

Notice how we use the `$simple{ }` syntax in the `toEndpoint` above. If you don't do this, there is a clash and Spring will throw an exception like

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.

To use SQL in your camel routes you need to add the a 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).

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.

And the spring DSL:



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.

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 <code>setVariables</code> 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:

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.

Namespaces

You can easily use namespaces with XPath expressions using the Namespaces helper class.



Streams

If the message body 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**. So often when you use XPath as Message Filter or Content Based Router then you need to access the data multiple times, and you should use Stream caching or convert the message body to a `String` prior which is safe to be re-read multiple times.

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>	in	Message	the exchange.in message
<code>http://camel.apache.org/xml/out/</code>	out	Message	the exchange.out message
<code>http://camel.apache.org/xml/function/</code>	functions	Object	Camel 2.5: Additional functions
<code>http://camel.apache.org/xml/variables/environment-variables</code>	env	Object	OS environment variables
<code>http://camel.apache.org/xml/variables/system-properties</code>	system	Object	Java System properties
<code>http://camel.apache.org/xml/variables/exchange-property</code>	Ê	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.

Notice: `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.

And the new functions introduced in Camel 2.5:

Using XML configuration

If you prefer to configure your routes in your Spring XML file then you can use XPath expressions as follows

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 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:

In Spring DSL you use the **resultType** attribute to provide a fully qualified classname:

In `@XPath`:

Available as of Camel 2.1

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 to a header's value you can do this by defining the 'headerName' attribute.

In XML DSL:

And in Java DSL you specify the headerName as the 2nd parameter as shown:

Examples

Here is a simple example using an XPath expression as a predicate in a Message Filter

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

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:

And the spring XML equivalent of the route:

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.

i.e. 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.

For example

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:

This will match the given predicate.

You can also evaluate for example as shown in the following three examples:

Evaluating with a String result is a common requirement and thus you can do it a bit simpler:

Using Saxon with XPathBuilder

Available as of Camel 2.3

You need to add **camel-saxon** as dependency to your project.

Its 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

Using ObjectModel

The easy one

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:

Camel will log at `INFO` level if it uses a non default XPathFactory such as:

To use Apache Xerces you can configure the system property

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

Specifying the object model

Shortcut

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:

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

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:

Spring DSL:

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:

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:

```
-----
```

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

Name	Default Value	Description
allowStAX	false	Camel 2.8.3/2.9: Whether to allow using StAX as the javax.xml.transform.Source.

Examples

```
-----
```

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.

```
-----
```

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

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:

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.

Examples

Here is a simple example using an XQuery expression as a predicate in a Message Filter

This example uses XQuery with namespaces as a predicate in a Message Filter

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:

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).

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
 - JSON
 - Protobuf
- Object/XML marshalling
 - Castor
 - JAXB
 - XmlBeans
 - XStream
 - JiBX
- Object/XML/Webservice marshalling
 - SOAP
- Direct JSON / XML marshalling
 - XmlJson
- Flat data structure marshalling
 - BeanIO
 - Bindy
 - CSV
 - EDI
 - Flatpack DataFormat
- Domain specific marshalling
 - HL7 DataFormat
- Compression
 - GZip data format
 - Zip DataFormat
 - Zip File DataFormat
- Security
 - Crypto
 - PGP
 - XMLSecurity DataFormat

- Misc.
 - Base64
 - Custom DataFormat - to use your own custom implementation
 - RSS
 - TidyMarkup
 - Syslog
 - ICal

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

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

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

Using Spring XML

This example shows how to configure the data type just once and reuse it on multiple routes

You can also define reusable data formats as Spring beans

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`

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`.

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.

Using Spring XML

The following example shows how to use JAXB to unmarshal using Spring configuring the *jaxb* data type

This example shows how to configure the data type just once and reuse it on multiple routes.

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 `@XmlElement` annotation on given class to operate on whole XML tree. This is useful



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.

but not always - sometimes generated code does not have `@XmlRootElement` annotation, sometimes you need unmarshall only part of tree. In that case you can use partial unmarshalling. To enable this behaviours you need set property `partClass`. Camel will pass this class to JAXB's unmarshaller.

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`.

	JDK 1.5	JDK 1.6+
Filtering in use	StAX API and implementation	No
Filtering not in use	StAX API only	No

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.

The following example shows using the Spring DSL and also enabling Camel's NonXML filtering:

Working with the ObjectFactory

If you use XJC 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:

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-R1 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.

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.

Schema validation

Available as of Camel 2.11

The JAXB Data Format supports validation by marshalling and unmarshalling from/to XML. You 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. Using the Java DSL, you can configure it in the following way:

You can do the same using the XML DSL:

Camel will create and pool the underlying `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:

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).

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.

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).

XSTREAM

XStream is a Data Format which uses the XStream library to marshal and unmarshal Java objects to and from XML.



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.

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.

```
-----
```

Dependencies

To use XStream in your camel routes you need to add the a 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).

```
-----
```

CSV

The CSV Data Format uses Apache Commons CSV to handle CSV payloads (Comma Separated Values) such as those exported/imported by Excel.

Options

Option	Type	Description
config	CSVConfig	Can be used to set a custom <code>CSVConfig</code> object.
strategy	CSVStrategy	Can be used to set a custom <code>CSVStrategy</code> ; the default is <code>CSVStrategy.DEFAULT_STRATEGY</code> .
autogenColumns	boolean	Whether or not columns are auto-generated in the resulting CSV. The default value is <code>true</code> ; 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 <code>,</code> .
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 <code>false</code> .

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.

An example: if you send a message with this map...

... through this route ...

... you will end up with a String containing this CSV message

```
abc,123
```

Sending the Map below through this route will result in a CSV message that looks like

```
foo,bar
```

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.

We can now use the CSV component to unmarshal this file:

The resulting message will contain a `List<List<String>>` like...

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

... your route then looks as follows

Marshaling with a pipe as delimiter

Using the Spring/XML DSL:

Or the Java DSL:

Using autogenColumns, configRef and strategyRef attributes inside XML DSL

Available as of Camel 2.9.2 / 2.10

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.

Using skipFirstLine option while unmarshaling

Available as of Camel 2.10

You can instruct the CSV Data Format to skip the first line which contains the CSV headers. Using the Spring/XML DSL:

Or the Java DSL:

Unmarshaling with a pipe as delimiter

Using the Spring/XML DSL:

Or the Java DSL:

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).

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.

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.

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 format between `String` and HL7 model objects.



Issue in CSVConfig

It looks like that

doesn't work. You have to set the delimiter as a String!

- `marshal` = from Message to byte stream (can be used when returning as response 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:

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:

Here we unmarshal the byte stream into a HAPI Message object that is passed to our patient lookup service.

Notice there is a shorthand syntax in Camel for well-known data formats that is commonly used.

Then you don't need to create an instance of the `HL7DataFormat` object:

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`.



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.



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()`).

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
<code>definition</code>	<code>null</code>	The flatpack pzmap configuration file. Can be omitted in simpler situations, but its preferred to use the pzmap.
<code>fixed</code>	<code>false</code>	Delimited or fixed.
<code>ignoreFirstRecord</code>	<code>true</code>	Whether the first line is ignored for delimited files (for the column headers).
<code>textQualifier</code>	<code>"</code>	If the text is qualified with a char such as <code>"</code> .
<code>delimiter</code>	<code>,</code>	The delimiter char (could be <code>;</code> , <code>,</code> or similar)
<code>parserFactory</code>	<code>null</code>	Uses the default Flatpack parser factory.
<code>allowShortLines</code>	<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    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:

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.

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).

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

By default Camel uses the XStream library.

Using JSON data format with 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 Jackson library

Using JSON data format with the GSON library

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.

And then you can refer to this id in the route:

Excluding POJO fields from marshallng

As of Camel 2.10

When marshalling 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.

Use the marker classes with the `@JsonView` annotation to include/exclude certain fields. The annotation also works on getters.

Finally use the Camel `JacksonDataFormat` to marshall the above POJO to JSON.

Note that the weight field is missing in the resulting JSON:

The GSON library supports a similar feature through the notion of `ExclusionStrategies`:

The `GsonDataFormat` accepts an `ExclusionStrategy` in its constructor:

The line above will exclude fields annotated with `@ExcludeAge` when marshalling 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

Which we want to map to a POJO that has getter/setters as

Then we can configure the `org.apache.camel.component.gson.GsonDataFormat` in a Spring XML files 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 pre defined 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.

And use it in Camel routes by referring to its bean id as shown:

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:

Directly specify your JSON view inside the Java DSL as:

And the same in XML DSL:

Dependencies for XStream

To use JSON in your camel routes you need to add the a 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).

Dependencies for Jackson

To use JSON in your camel routes you need to add the a 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).

Dependencies for GSON

To use JSON in your camel routes you need to add the a 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).

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.

Options

Option	Default	Description
compressionLevel	null	<p>To specify a specific compression Level use <code>java.util.zip.Deflater</code> settings. The possible settings are</p> <p>0 - Deflater.BEST_SPEED</p> <p>1 - Deflater.BEST_COMPRESSION</p> <p>9 - Deflater.DEFAULT_COMPRESSION</p> <p>If compressionLevel is not explicitly specified the compressionLevel 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`.

Alternatively if you would like to use the default setting you could send it as



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.

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.

Dependencies

This data format is provided in **camel-core** so no additional dependencies is 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.

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

Spring XML Example

The following example shows how to use TidyMarkup to unmarshal using Spring

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).



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.

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

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.

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),



Format supported

This first release only support comma separated values fields and key value pair fields (e.g. : FIX messages).



Multiple models

If you use multiple models, each model has to be placed in it's own package to prevent unpredictable results.

- Section (to identify header, body and footer section),
- OneToMany

This section will describe them :

I. 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

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
Ê	Ê	This annotation is associated to the root class of the model and must be declared one time.

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

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

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

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 would 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"

From **Camel 2.8.3/2.9 or newer** bindy will automatic detect if the record is enclosed with either single or double quotes and automatic remove those quotes when unmarshalling from CSV to Object. Therefore do **not** include the quotes in the separator, but simple do as below:

"10","J","Pauline"," M","XD12345678","Fortis Dynamic 15,15" 2500","USD","08-01-2009"

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:

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 :

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 :

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

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:

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.

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	
Ê	Ê	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

AND for the class Client :

Listing 1. Class Link

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 - digit number starting from 1 to ...
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 Formater (SimpleDateFormat by example) to format/ validate data
position	int	optional - must be used when the position of the field in the CSV generated must be different compare to pos
required	boolean	optional - default value = "false"
trim	boolean	optional - default value = "false"
defaultValue	string	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

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

case 2 : pattern

The pattern allows to enrich or validates the format of your data

Listing 1. Pattern

case 3 : precision

The precision is helpful when you want to define the decimal part of your number

Listing 1. Precision

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

case 5 : required

If a field is mandatory, simply use the attribute 'required' setted to true

Listing 1. Required

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



This attribute of the annotation `@DataField` must be used in combination with attribute `isOrdered = true` of the annotation `@CsvRecord`

case 7 : defaultValue

If a field is not defined then uses the value indicated by the `defaultValue` attribute

Listing 1. Default value

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 beause 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).



This attribute is only applicable to optional fields.

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 field can be ignored when unmarshalling / parsing.
Ê	Ê	This annotation is associated to the root class of the model and must be declared one time.

case 1 : Simple fixed length record

This simple example shows how to design the model to parse/format a fixed message

```
I0A9PaulineMISINXD I2345678BUYShare2500.45USD01-08-2009
```

Listing 1. Fixed-simple

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"

```
I0A9 PaulineM ISINXD I2345678BUYShare2500.45USD01-08-2009
```

Listing 1. Fixed-padding-align

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 padd with '0' instead of ' '. In this case, you can use in the model the attribute paddingField to set this value.

```
I0A9 PaulineM ISINXD I2345678BUYShare000002500.45USD01-08-2009
```

Listing 1. Fixed-padding-field

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:

```
I0A9Pauline^M^ISINXD I2345678BUYShare000002500.45USD01-08-2009
```



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.

Listing 1. Fixed-delimited

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

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

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 accomodate 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.

5. Message

The Message annotation is used to identified 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).



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.

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)

Annotation name	Record type	Level
Message	key value pair	Class

Parameter name	type	Info
pairSeparator	string	mandatory - can be '=' or ';' or 'anything'
keyValuePairSeparair	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
Ê	Ê	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=I 35=0 49=INVMGR 56=BRKR I=BE.CHM.001 I I=CHM0001-01 22=4
```

...

and how to use the annotation

Listing 1. FIX - message



"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/>.



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
KeyValuePairField	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

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

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
Section	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

B. Definition of the body section

Listing 1. FIX message - Section - Body

C. Definition of the footer section

Listing 1. FIX message - Section - Footer

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.

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
OneToMany	all	property



Restrictions OneToMany

Be careful, the one to many of bindy does not allow to handle repetitions defined on several levels of the hierarchy

Parameter name	type	Info
mappedTo	string	optional - string - class name associated to the type of the List<Type of the Class>

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

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=INVMGR56=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

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.

Unmarshaling

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:

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.

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:

Marshaling

To generate CSV records from a collection of model objects, you create the following route :

Unit test

Here is two examples showing how to marshal or unmarshal a CSV file with Camel

Listing 1. Marshall

Listing 1. Unmarshall

In this example, `BindyCsvDataFormat` class has been instantiated in a traditional way but it is also possible to provide information directly to the function (un)marshal like this where `BindyType` corresponds to the `Bindy DataFormat` class to instantiate and the parameter contains the list of package names.

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

Listing 1. spring dsl

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).

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.

**Be careful**

Please verify that your model classes implements serializable otherwise the queue manager will raise an error

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 Leveltrue = 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 passPhase 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.TRIPLEDESXMLCipher.AES_128XMLCipher.AES_128_GCM Camel 2.12 <ul style="list-style-type: none">XMLCipher.AES_192XMLCipher.AES_192_GCM Camel 2.12 <ul style="list-style-type: none">XMLCipher.AES_256XMLCipher.AES_256_GCM Camel 2.12

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 a public or private key from a KeyStore instance performing asymmetric key encryption.
keyCipherAlgorithm	Camel 2.12 XMLCipher.RSA_OAEP	The cipher algorithm to be used for the encryption and decryption of the asymmetric key. The available choices are: <ul style="list-style-type: none"> XMLCipher.RSA_V1_5 XMLCipher.RSA_OAEP XMLCipher.RSA_OAEP_MGF1
keyOrTrustStoreParameters	null	Configuration options for creating a KeyStore instance that represents the sender's trustStore or recipient's keyStore.
keyPassword	null	Camel 2.10.2 / 2.11: The password used for retrieving the private key from the KeyStore. It is used for asymmetric decryption.
digestAlgorithm	XMLCipher.SHA1	Camel 2.12 The digest algorithm to be used for 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 be used for the RSA OAEP algorithm. The available choices are: <ul style="list-style-type: none"> EncryptionConstants.MGF1_SHA1 EncryptionConstants.MGF1_SHA256 EncryptionConstants.MGF1_SHA512

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

Partial Payload Content Only encryption/decryption

Partial Multi Node Payload Content Only encryption/decryption

Partial Payload Content Only encryption/decryption with choice of `passPhrase(password)`

Partial Payload Content Only encryption/decryption with passPhrase(password) and Algorithm

Partial Payload Content with Namespace support

Java DSL

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.

Asymmetric Key Encryption

Spring XML Sender

Spring XML Recipient

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.

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`.

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

For example the following uses a named DataFormat of Castor which uses default Castor data binding features.

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.

If you want to override default mapping schema by providing a mapping file you can set it as follows.

Also if you want to have more control on Castor Marshaller and Unmarshaller you can access them as below.

Using Spring XML

The following example shows how to use Castor to unmarshal using Spring configuring the castor data type

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>`.

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).

Protobuf - Protocol Buffers

"Protocol Buffers - Google's data interchange format"

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



Available from Camel 2.2

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
```

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 is 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.

Or use the DSL protobuf() passing the unmarshal default instance or default instance class name like this.

SPRING DSL

The following example shows how to use Castor to unmarshal using Spring configuring the protobuf data type

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).

```
<code></code>
```

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.

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.)

```
<code></code>
```



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.



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

When using XML DSL there is a version attribute you can set on the <soap> element.

And in the Camel 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.

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.

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.

The below snippet creates a proxy for the service interface and makes a SOAP call to the above route.

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.

Dependencies

To use the SOAP dataformat in your camel routes you need to add the following dependency to your pom.

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 marshal and unmarshal formatting mechanism. It assumes marshalling to mean encryption to cyphertext and unmarshalling to mean decryption back to the original plaintext.

Options

Name	Type	Default	Description
algorithm	String	DES/CBC/ PKCS5Padding	The JCE algorithm name indicating the cryptographic algorithm that will be used.
algorithmParameterSpec	AlgorithmParameterSpec	null	A JCE AlgorithmParameterSpec used to initialize the Cipher.
bufferSize	Integer	2048	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 the configured IV should be inlined into the encrypted data stream.
macAlgorithm	String	null	The JCE algorithm name indicating the Message Authentication algorithm.
shouldAppendHMAC	boolean	null	Flag indicating that a Message Authentication Code should be calculated and appended to the encrypted data.

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:

In Spring the dataformat is configured first and then used in routes

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.

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.

or with spring, supplying a reference to a `byte[]`

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.

or with spring.

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 <http://en.wikipedia.org/wiki/HMAC>

or with spring.

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

or with spring.

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"`

or with spring.

PGPDataFormat Options

Name	Type	Default	Description
keyUserId	String	null	The userid of the key in the PGP keyring.
password	String	null	Password used when opening the private key (not used for encryption).
keyFileName	String	null	Filename of the keyring; must be accessible as a classpath resource (but you can specify a location in the file system by using the "file:" prefix).
encryptionKeyRing	byte[]	null	Since camel 2.12.1 encryption keyring; you can not set the keyFileName and encryptionKeyRing at the same time.
signatureKeyUserId	String	null	Since Camel 2.11.0 Optional userid of the key in the PGP keyring to use for signing (during encryption) or signature verification (during decryption) .
signaturePassword	String	null	Since Camel 2.11.0 Optional password used when opening the private key used for signing (during encryption).
signatureKeyFileName	String	null	Since Camel 2.11.0 Optional filename of the keyring to use for signing (during encryption) or for signature verification (during decryption); must be accessible as a classpath resource (but you can specify a location in the file system by using the "file:" prefix).
signatureKeyRing	byte[]	null	Since camel 2.12.1 signature keyring; you can not set the signatureKeyFileName and signatureKeyRing at the same time.
armored	boolean	false	This option will cause PGP to base64 encode the encrypted text, making it available for copy/paste, etc.
integrity	boolean	true	Adds an integrity check/sign into the encryption file.

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 userid 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 userid of the signature key in the PGP keyring; will override existing setting directly on the PGPDataFormat.

CamelPGPDataFormatSignatureKeyPassword String

Since Camel 2.11.0 Password used when opening the signature private key; will override existing setting directly on the PGPDataFormat.

Encrypting with PGPDataFormat

The following sample uses the popular PGP format for encrypting/decrypting files using the Bouncy Castle Java libraries:

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:

Or using Spring:

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

2. Create your keyring, entering a secure password
[]
3. If you need to import someone else's public key so that you can encrypt a file for them.
[]
4. The following files should now exist and can be used to run the example
[]

Dependencies

To use the Crypto dataformat in your camel routes you need to add the following dependency to your pom.

[]

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 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.

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

[]

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

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:

The same route using camel-mina

Sending syslog messages to a remote destination

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

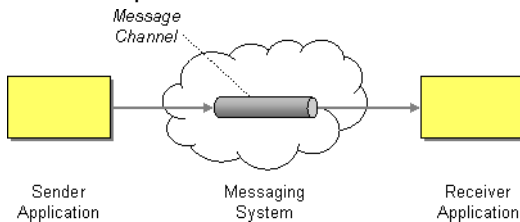
Pattern Appendix

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

MESSAGING SYSTEMS

Message Channel

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



For more details see

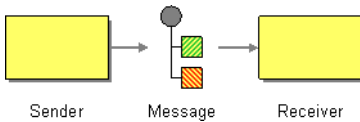
- Message
- Message Endpoint

Using This Pattern

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

Message

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



To support various message exchange patterns like one way Event 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

Route Using the Fluent Builders

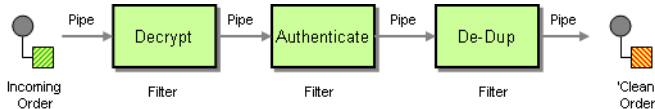
Route Using the Spring XML Extensions

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

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

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

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

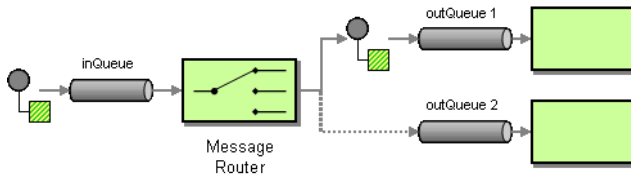
In the above example we are routing from a single Endpoint to a list of different endpoints specified using URIs. If you find the above a bit confusing, try reading about the Architecture or try the Examples

Using This Pattern

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

Message Router

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



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

Using the Fluent Builders

Using the Spring XML Extensions

Choice without otherwise

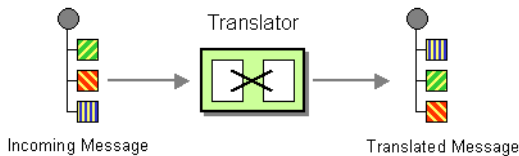
If you use a `choice` without adding an `otherwise`, any unmatched exchanges will be dropped by default.

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

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

or you can use the DSL to explicitly configure the transformation

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

Or you can use the Bean Integration to invoke a bean

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)

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.

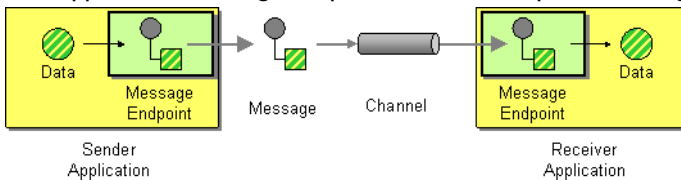
Using This Pattern

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

- Content 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. It's then a responsibility of the CamelContext to create and activate the necessary Endpoint instances using the available Component implementations.

For more details see

- Message

Using This Pattern

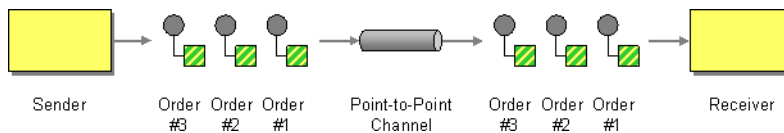
If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

MESSAGING CHANNELS

Point to Point Channel

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

- 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



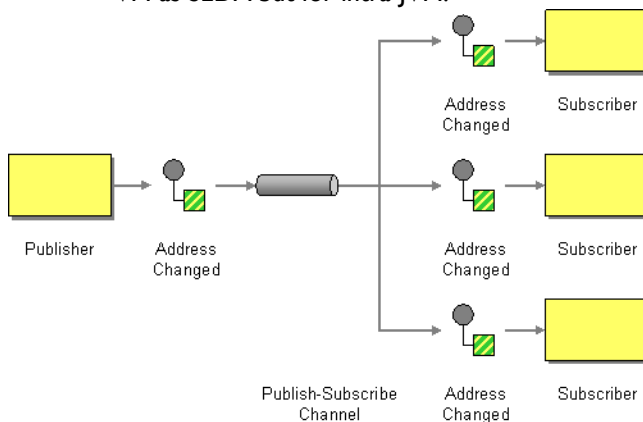
Using This Pattern

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

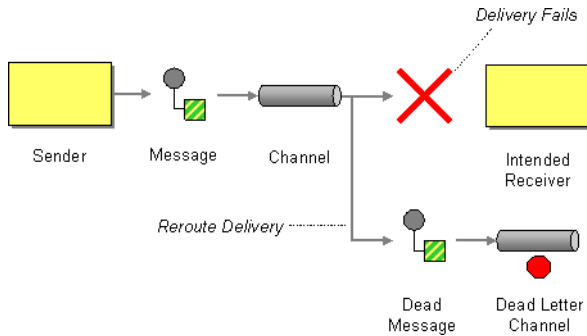
Using the Spring XML Extensions

Using This Pattern

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

DEAD LETTER CHANNEL

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



Redelivery

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

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

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



Difference between Dead Letter Channel and Default Error Handler

The major difference is that Dead Letter Channel has a dead letter queue that whenever an Exchange could not be processed is moved to. It will **always** move failed exchanges to this queue.

Unlike the Default Error Handler that does **not** have a dead letter queue. So whenever an Exchange could not be processed the error is propagated back to the client.

Notice: You can adjust this behavior of whether the client should be notified or not with the **handled** option.

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

Using the Spring XML Extensions

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:

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:

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.

Redelivery default values

Redelivery is disabled by default.

The default redeliver policy will use the following values:

- `maximumRedeliveries=0`
- `redeliverDelay=1000L` (1 second)
- `maximumRedeliveryDelay = 60 * 1000L` (60 seconds)
- And the exponential backoff and collision avoidance is turned off.
- The `retriesExhaustedLogLevel` are set to `LogLevel.ERROR`
- The `retryAttemptedLogLevel` are set to `LogLevel.DEBUG`
- Stack traces is logged for exhausted messages from Camel 2.2 onwards.
- Handled exceptions is not logged from Camel 2.3 onwards

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

The maximum redeliveries 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` to a negative value such as -1 will then always redelivery (unlimited).

Setting the `maximumRedeliveries` to 0 will disable any re delivery attempt.

Camel will log delivery failures at the `DEBUG` logging level by default. You can change this by specifying `retriesExhaustedLogLevel` and/or `retryAttemptedLogLevel`. See `ExceptionHandlerWithRetryLogLevelSetTest` for an example.

You can turn logging of stack traces on/off. If turned off Camel will still log the redelivery attempt. Its just much less verbose.

Redeliver Delay Pattern

Delay pattern is used as a single option to set a range pattern for delays. If used then the following options does not apply: (`delay`, `backOffMultiplier`, `useExponentialBackOff`, `useCollisionAvoidance`, `maximumRedeliveryDelay`).



onException and onRedeliver

We also support for per **onException** to set a **onRedeliver**. 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.

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 millis

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 3 groups:

- 5:1000
- 10:5000
- 20:20000

Resulting in these delays for redelivery attempt:

- Redelivery attempt number 1..4 = 0 millis (as the first group start with 5)
- Redelivery attempt number 5..9 = 1000 millis (the first group)
- Redelivery attempt number 10..19 = 5000 millis (the second group)
- Redelivery attempt number 20.. = 20000 millis (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 eg have a starting delay:

```
delayPattern=1:1000;5:5000
```

- Redelivery attempt number 1..4 = 1000 millis (the first group)
- Redelivery attempt number 5.. = 5000 millis (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`.

Starting with 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`.
Is 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:

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:

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 useable if the Camel route is a bit dynamic such as the dynamic Recipient List so you know which endpoints failed.

Notice: These information is kept on the Exchange even if the message was successfully processed by a given endpoint, and then later fails for example in a local Bean processing instead. So beware that this is a hint that helps pinpoint errors.

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`.

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.

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

Prior to Camel 2.10, Camel will perform redelivery while stopping a route, or shutting down Camel. This has improved a bit in Camel 2.10 onwards, as Camel will not perform redelivery attempts when shutting down aggressively (eg during Graceful Shutdown and timeout hit). From Camel 2.11 onwards 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` to be backwards compatible as before. For example the following sample shows how to do this with Java DSL and XML DSL

And the sample sample with XML DSL

Samples

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

You can also configure the RedeliveryPolicy as this example shows

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 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.

Here we configure the Dead Letter Channel to use our processor `MyRedeliveryProcessor` to be executed before each redelivery.

And this is the processor `MyRedeliveryProcessor` where we alter the message.

Using This Pattern

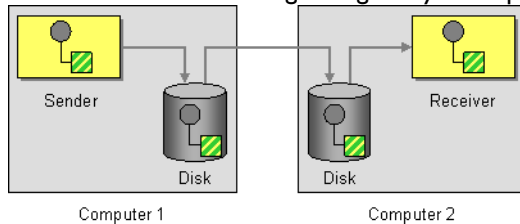
If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

- 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

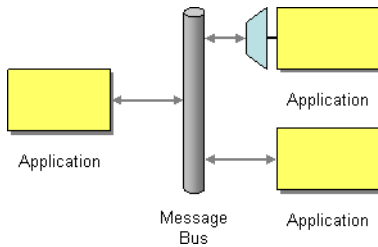


Using This Pattern

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.

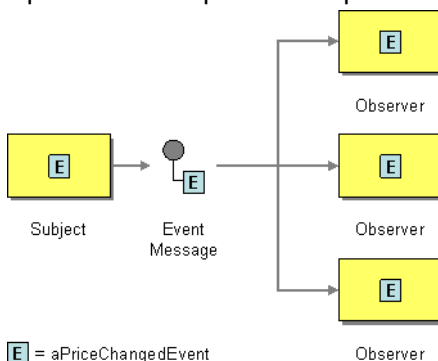
Using This Pattern

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.

From 2.0 onwards on Camel you can specify the Exchange Pattern using the dsl.

Using the Fluent Builders

or you can invoke an endpoint with an explicit pattern

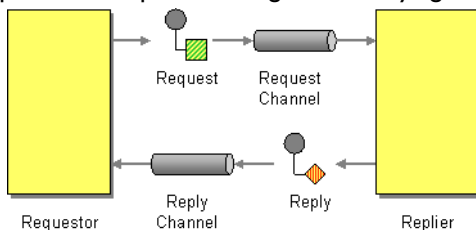
Using the Spring XML Extensions

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).

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.

You can specify the exchange pattern in DSL rule or Spring configuration.

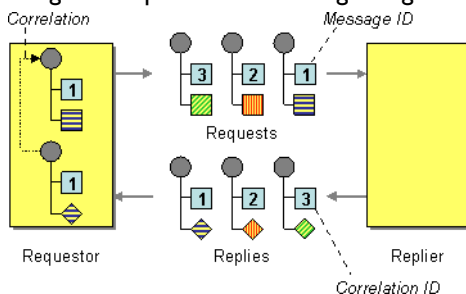
Using This Pattern

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 the key `Exchange.CORRELATION_ID`, which links back to the source Exchange. For example the Splitter, Multicast, Recipient List, and Wire Tap EIP does this.



Related

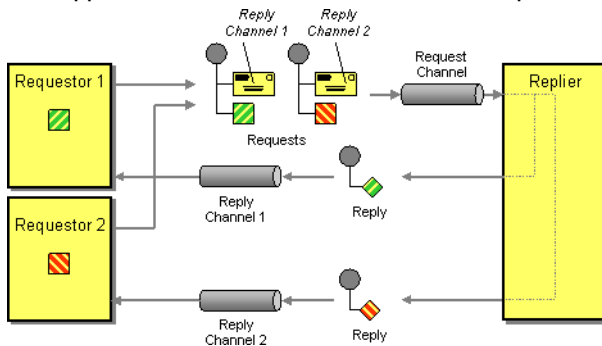
See the related Event Message message

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

Route Using the Fluent Builders

Route Using the Spring XML Extensions

For a complete example of this pattern, see this [junit test case](#)

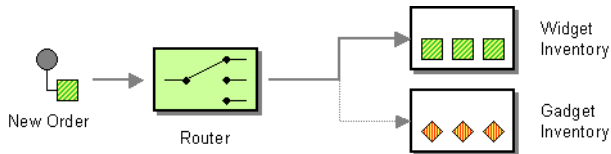
Using This Pattern

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

Using the Spring XML Extensions

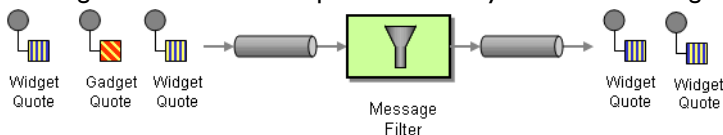
For further examples of this pattern in use you could look at the junit test case

Using This Pattern

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

Message 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

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

Here is another example of using a bean to define the filter behavior



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

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

Using stop

Available as of Camel 2.0

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()`.

Knowing if Exchange was filtered or not

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

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.

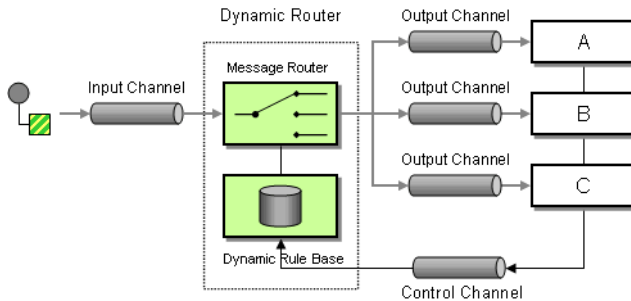


filtered endpoint required inside </filter> tag

make sure you put the endpoint you want to filter (<to uri="seda:b"/>, etc.) before the closing </filter> tag or the filter will not be applied (in 2.8+, omitting this will result in an error)

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*.

Options

Name	Default Value	Description
<code>uriDelimiter</code>	<code>,</code>	Delimiter used if the Expression returned multiple endpoints.
<code>ignoreInvalidEndpoints</code>	<code>false</code>	If an endpoint uri could not be resolved, should it be ignored. Otherwise Camel will thrown an exception stating the endpoint uri is not valid.

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:

```

[.....]

```



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.

Which will leverage a Bean to compute the slip *on-the-fly*, which could be implemented as follows:

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:

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.

Spring XML

The same example in Spring XML would be:

@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.

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.

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



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.

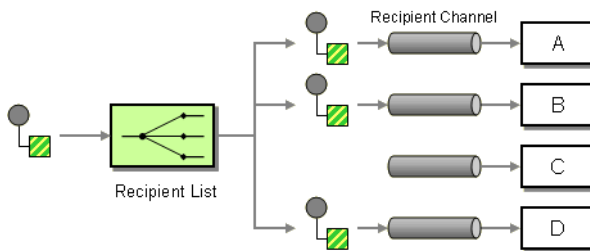
- Bean component

Using This Pattern

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

Recipient List

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



The recipients will receive a copy of the **same** Exchange, and Camel will execute them sequentially.

Options

Name	Default Value	Description
delimiter	,	Delimiter used if the Expression returned multiple endpoints.
strategyRef	Ê	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 Aggregate page for more details.
strategyMethodName	Ê	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	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's the sending and processing of replies from recipients which happens in parallel.
executorServiceRef	Ê	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 <code>AggregationStrategy</code> 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.
timeout	Ê	Camel 2.5: Specifies a processing timeout 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 <code>TimeoutAwareAggregationStrategy</code> , its <code>timeout</code> 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 a bit of care. We may be able to improve this functionality in future Camel releases.
onPrepareRef	Ê	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.

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

Using the Spring XML Extensions

Dynamic Recipient List

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

Using the Fluent Builders

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

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 with comma
- any other type will be regarded as a single value

Using the Spring XML Extensions

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

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 customer delimiter to use instead.

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 archive the same. The route above in Java DSL:

In **Camel 2.1** its a bit easier as you can pass in the delimiter as 2nd parameter:

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.

And in Spring XML its an attribute on the recipient list tag.

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.

And in Spring XML its an attribute on the recipient list tag.

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` which the Routing Slip also supports. You can use it to skip endpoints which is invalid.

And in Spring XML its an attribute on the recipient list tag.

Then lets say the `myHeader` contains the following two endpoints `direct:foo,xxx:bar`. The first endpoint is valid and works. However the 2nd is invalid and will just be ignored. Camel logs at INFO level about, so you can see why the endpoint was invalid.

Using custom AggregationStrategy

Available as of Camel 2.2

You can now use you own `AggregationStrategy` with the Recipient List. However its not that often you need that. What its good for is that in case you are using Request Reply messaging then the replies from the recipient can be aggregated. By default Camel uses `UseLatestAggregationStrategy` which just keeps that last received reply. What if you must remember all the bodies that all the recipients send back, then you can use your own custom aggregator that keeps those. Its the same principle as with the Aggregator EIP so check it out for details.

And in Spring XML its an attribute on the recipient list tag.

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 the sequential order (also if parallel processing is enabled) of the endpoints the Recipient List is using. However from Camel 2.12 this is easier to know as the `newExchange` `Exchange` has a property stored (key is `Exchange.RECIPIENT_LIST_ENDPOINT` with the uri of the Endpoint.

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 a future version).

Using method call as recipient list

You can use a Bean to provide the recipients, for example:

And then `MessageRouter`:

When you use a Bean then do **not** also use the `@RecipientList` annotation as this will in fact add yet another recipient list, so you end up having two. Do **not** do like this.

Well you should only do like that above (using `@RecipientList`) if you route just route to a Bean which you then want to act as a recipient list.

So the original route can be changed to:

Which then would invoke the `routeTo` method and detect its 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 is slow. For example you can set a timeout value of 20 sec. For example in the unit test below you can see 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".

By default if a timeout occurs the `AggregationStrategy` is not invoked. However you can implement a specialized version

This allows you to deal with the timeout in the `AggregationStrategy` if you really need to.



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.



Timeout in other EIPs

This `timeout` feature is also supported by Splitter and both `multicast` and `recipientList`.



Timeout is total

The timeout is total, which means that after X time, Camel will aggregate the messages which has 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

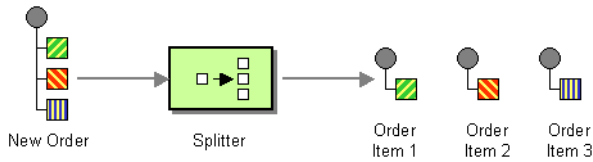
See details at [Multicast](#)

Using This Pattern

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

Splitter

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



You need to specify a Splitter as `split()`. In earlier versions of Camel, you need to use `splitter()`.

Options

Name	Default Value	Description
<code>strategyRef</code>	<code>Ê</code>	Refers to an <code>AggregationStrategy</code> 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 <code>AggregationStrategy</code> , see the <i>Aggregate</i> page for more details.
<code>strategyMethodName</code>	<code>Ê</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 <i>Aggregate</i> page for more details.
<code>strategyMethodAllowNull</code>	<code>false</code>	Camel 2.12: If this option is <code>false</code> then the <code>aggregate</code> method is not used for the very first splitted message. If this option is <code>true</code> then <code>null</code> values is used as the <code>oldExchange</code> (for the very first message splitted), when using POJOs as the <code>AggregationStrategy</code> . See the <i>Aggregate</i> page for more details.
<code>parallelProcessing</code>	<code>false</code>	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.
<code>executorServiceRef</code>	<code>Ê</code>	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.
<code>stopOnException</code>	<code>false</code>	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 <code>AggregationStrategy</code> class where you have full control how to handle that.
<code>streaming</code>	<code>false</code>	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 where splitted.
<code>timeout</code>	<code>Ê</code>	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 <code>TimeoutAwareAggregationStrategy</code> then the <code>timeout</code> 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.
<code>onPrepareRef</code>	<code>Ê</code>	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.
<code>shareUnitOfWork</code>	<code>false</code>	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
<code>CamelSplitIndex</code>	<code>int</code>	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 **queue:a** endpoint the split it into pieces using an Expression, then forward each piece to **queue:b**

Using the Fluent Builders

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.

Using the Spring XML Extensions

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

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**.

Splitting the body in Spring XML is a bit harder as you need to use the Simple language to dictate this

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.

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`.

Stream based

You can split streams by enabling the streaming mode using the `streaming` builder method.

You can also supply your custom splitter to use with streaming like this:

Streaming big XML payloads using Tokenizer language

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.

For example you may have a XML payload structured as follows

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:

In XML DSL the route would be as follows:

Notice the `tokenizeXML` method which will split the file using the tag name of the child node, 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:

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:

And in Java DSL its as follows:

Splitting files by grouping N lines together

Available as of Camel 2.10



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.



StAX component

The Camel StAX component can also be used to split big XML files in a streaming mode. See more details at StAX.

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.

And in XML DSL

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:

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.

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.

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.

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:

And the OrderService bean is as follows:

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.

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:

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 thrown 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:



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.

And using XML DSL you specify it as follows:

```
-----
```

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

```
-----
```

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:

```
-----
```



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.

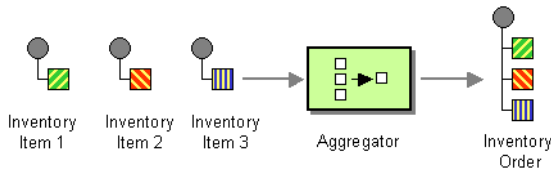
Using This Pattern

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.

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:

Option	Default	Description
<code>correlationExpression</code>	<code>Ê</code>	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 <code>ignoreBadCorrelationKeys</code> option.
<code>aggregationStrategy</code>	<code>Ê</code>	Mandatory <code>AggregationStrategy</code> which is used to <i>merge</i> the incoming Exchange with the existing already merged exchanges. At first call the <code>oldExchange</code> parameter is null. On subsequent invocations the <code>oldExchange</code> contains the merged exchanges and <code>newExchange</code> is of course the new incoming Exchange. From Camel 2.9.2 onwards the strategy can also be a <code>TimeoutAwareAggregationStrategy</code> implementation, supporting the timeout callback, see further below for more details.
<code>strategyRef</code>	<code>Ê</code>	A reference to lookup the <code>AggregationStrategy</code> in the Registry. From Camel 2.12 onwards you can also use a POJO as the <code>AggregationStrategy</code> , see further below for details.
<code>strategyMethodName</code>	<code>Ê</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 further below for more details.

strategyMethodAllowNull	false	Camel 2.12: If this option is <code>false</code> then the <code>aggregate</code> method is not used for the very first aggregation. If this option is <code>true</code> then <code>null</code> values is used as the <code>oldExchange</code> (at the very first aggregation), when using POJOs as the <code>AggregationStrategy</code> . See further below for more details.
completionSize	Ê	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 <code>Integer</code> as result. If both are set Camel will fallback to use the fixed value if the Expression result was <code>null</code> or 0.
completionTimeout	Ê	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 <code>Long</code> as result. If both are set Camel will fallback to use the fixed value if the Expression result was <code>null</code> or 0. You cannot use this option together with <code>completionInterval</code> , only one of the two can be used.
completionInterval	Ê	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 <code>completionTimeout</code> , only one of them can be used.
completionPredicate	Ê	A Predicate to indicate when an aggregated exchange is complete.
completionFromBatchConsumer	false	This option is if the exchanges are coming from a Batch Consumer. Then when enabled the <code>Aggregator2</code> will use the batch size determined by the Batch Consumer in the message header <code>CamelBatchSize</code> . 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
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 <code>completionPredicate</code> option as the Exchange being passed in changes accordingly. When <code>false</code> 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 <code>AggregationStrategy</code> is available for the Predicate. When <code>true</code> 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 <code>org.apache.camel.impl.GroupedExchange</code> 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 <code>AggregationStrategy</code> yourself. Important: 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	Ê	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 <code>closedCorrelationKeyException</code> exception. When using this option you pass in a integer which is a number for a LRU Cache which keeps that last X number of closed correlation keys. You can pass in 0 or a negative value to indicate a unbounded cache. By passing in a number you are ensured that cache won't grow too big if you use a log 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	Ê	Allows you to plugin you own implementation of <code>org.apache.camel.spi.AggregationRepository</code> which keeps track of the current inflight aggregated exchanges. Camel uses by default a memory based implementation.
aggregationRepositoryRef	Ê	Reference to lookup a <code>aggregationRepository</code> 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	Ê	If using <code>parallelProcessing</code> you can specify a custom thread pool to be used. In fact also if you are not using <code>parallelProcessing</code> this custom thread pool is used to send out aggregated exchanges as well.
executorServiceRef	Ê	Reference to lookup a <code>executorService</code> in the Registry
timeoutCheckerExecutorService	Ê	Camel 2.9: If using either of the <code>completionTimeout</code> , <code>completionTimeoutExpression</code> , or <code>completionInterval</code> 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	Ê	Camel 2.9: Reference to lookup a <code>timeoutCheckerExecutorService</code> in the Registry
optimisticLocking	false	Camel 2.11: Turns on using optimistic locking, which requires the <code>aggregationRepository</code> being used, is supporting this by implementing the <code>org.apache.camel.spi.OptimisticLockingAggregationRepository</code> interface.
optimisticLockRetryPolicy	Ê	Camel 2.11.1: Allows to configure retry settings when using optimistic locking.

Exchange Properties

The following properties are set on each aggregated Exchange:

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, consumer, timeout 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.

[]

About completion

When aggregation Exchanges at some point you need to indicate that the aggregated exchanges is complete, so they can be send out of the aggregator. Camel allows you to indicate completion in various ways as follows:

- `completionTimeout` - Is an inactivity timeout in which is triggered if no new exchanges have been aggregated for that particular correlation key within the period.
- `completionInterval` - Once every X period all the current aggregated exchanges are completed.
- `completionSize` - Is a number indicating that after X aggregated exchanges it's complete.
- `completionPredicate` - Runs a Predicate when a new exchange is aggregated to determine if we are complete or not
- `completionFromBatchConsumer` - Special option for Batch Consumer which allows you to complete when all the messages from the batch has been aggregated.
- `forceCompletionOnStop` - **Camel 2.9** Indicates to complete all current aggregated exchanges when the context is stopped

Notice that all the completion ways are per correlation key. And you can combine them in any way you like. It's basically the first which triggers that wins. So you can use a completion size together with a completion timeout. Only `completionTimeout` and `completionInterval` cannot be used at the same time.

Notice the completion is a mandatory option and must be provided to the aggregator. If not provided Camel will throw an Exception on startup.



Callbacks

See the `TimeoutAwareAggregationStrategy` and `CompletionAwareAggregationStrategy` extensions to `AggregationStrategy` that has callbacks when the aggregated Exchange was completed and if a timeout occurred.

Persistent AggregationRepository

The aggregator provides a pluggable repository which you can implement your own `org.apache.camel.spi.AggregationRepository`.

If you need persistent repository then you can use either Camel HawtDB, LevelDB, or SQL Component components.

Examples

See some examples from the old Aggregator which is somewhat similar to this new aggregator.

Using completionTimeout

In this example we want to aggregate all incoming messages and after 3 seconds of inactivity we want the aggregation to complete. This is done using the `completionTimeout` option as shown:

```
aggregation {
    completionTimeout(3000)
}
```

And the same example using Spring XML:

```
<aggregation completionTimeout="3000">
```

Using TimeoutAwareAggregationStrategy

Available as of Camel 2.9.2

If your aggregation strategy implements `TimeoutAwareAggregationStrategy`, then Camel will invoke the `timeout` method when the timeout occurs. Notice that the values for `index` and `total` parameters will be `-1`, and the `timeout` parameter will be provided only if configured as a fixed value. You must **not** throw any exceptions from the `timeout` method.

Using CompletionAwareAggregationStrategy

Available as of Camel 2.9.3

If your aggregation strategy implements `CompletionAwareAggregationStrategy`, then Camel will invoke the `onComplete` method when the aggregated Exchange is completed.



Setting options in Spring XML

Many of the options are configurable as attributes on the `<aggregate>` tag when using Spring XML.

This allows you to do any last minute custom logic such as to cleanup some resources, or additional work on the exchange as it's now completed.

You must **not** throw any exceptions from the `onCompletion` method.

Using completionSize

In this example we want to aggregate all incoming messages and when we have 3 messages aggregated (in the same correlation group) we want the aggregation to complete. This is done using the `completionSize` option as shown:

And the same example using Spring XML:

Using completionPredicate

In this example we want to aggregate all incoming messages and use a Predicate to determine when we are complete. The Predicate can be evaluated using either the aggregated exchange (default) or the incoming exchange. We will do both situations as examples. We start with the default situation as shown:

And the same example using Spring XML:

And the other situation where we use the `eagerCheckCompletion` option to tell Camel to use the incoming Exchange. Notice how we can just test in the completion predicate that the incoming message is the *END* message:

And the same example using Spring XML:

Using dynamic completionTimeout

In this example we want to aggregate all incoming messages and after a period of inactivity we want the aggregation to complete. The period should be computed at runtime based on the `timeout` header in the incoming messages. This is done using the `completionTimeout` option as shown:

And the same example using Spring XML:

Note: You can also add a fixed timeout value and Camel will fallback to use this value if the dynamic value was `null` or `0`.

Using dynamic completionSize

In this example we want to aggregate all incoming messages based on a dynamic size per correlation key. The size is computed at runtime based on the `mySize` header in the incoming messages. This is done using the `completionSize` option as shown:

And the same example using Spring XML:

Note: You can also add a fixed size value and Camel will fallback to use this value if the dynamic value was `null` or `0`.

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.

Manually Force the Completion of All Aggregated Exchanges Immediately

Available as of Camel 2.9

You can manually trigger completion of all current aggregated exchanges by sending a message containing the header `Exchange.AGGREGATION_COMPLETE_ALL_GROUPS` set to `true`. The message is considered a signal message only, the message headers/contents will not be processed otherwise.

Available as of Camel 2.11

You can alternatively set the header `Exchange.AGGREGATION_COMPLETE_ALL_GROUPS_INCLUSIVE` to `true` to trigger completion of all groups after processing the current message.

Using a List<V> in AggregationStrategy

Available as of Camel 2.11

If you want to aggregate some value from the messages `<V>` into a `List<V>` then we have added a

`org.apache.camel.processor.aggregate.AbstractListAggregationStrategy` abstract class in **Camel 2.11** that makes this easier. The completed Exchange that is sent out of the aggregator will contain the `List<V>` in the message body.

For example to aggregate a `List<Integer>` you can extend this class as shown below, and implement the `getValue` method:

```
-----
```

Using GroupedExchanges

In the route below we group all the exchanges together using `groupExchanges()`:

```
-----
```

As a result we have one outgoing Exchange being routed to the "mock:result" endpoint. The exchange is a holder containing all the incoming Exchanges.

To get access to these exchanges you need to access them from a property on the outgoing exchange as shown:

```
-----
```

From **Camel 2.13** onwards this behavior has changed to store these exchanges directly on the message body which is more intuitive:

```
-----
```

Using POJOs as AggregationStrategy

Available as of Camel 2.12

To use the `AggregationStrategy` you had to implement the

`org.apache.camel.processor.aggregate.AggregationStrategy` interface, which means your logic would be tied to the Camel API. From **Camel 2.12** onwards you can use a POJO for the logic and let Camel adapt to your POJO. To use a POJO a convention must be followed:

- there must be a public method to use
- the method must not be void
- the method can be static or non-static
- the method must have 2 or more parameters
- the parameters is paired so the first 50% is applied to the `oldExchange` and the remainder 50% is for the `newExchange`
- .. meaning that there must be an equal number of parameters, eg 2, 4, 6 etc.

The paired methods is expected to be ordered as follows:

- the first parameter is the message body
- the 2nd parameter is a Map of the headers
- the 3rd parameter is a Map of the Exchange properties

This convention is best explained with some examples.

In the method below, we have only 2 parameters, so the 1st parameter is the body of the `oldExchange`, and the 2nd is paired to the body of the `newExchange`:

```
-----
```



Notice the old way using the property is still present in **Camel 2.13** onwards, but its considered deprecated and to be removed in Camel 3.0 onwards.



You can use POJOs as `AggregationStrategy` with the other EIPs that supports aggregation, such as Splitter, Recipient List, etc.

In the method below, we have only 4 parameters, so the 1st parameter is the body of the `oldExchange`, and the 2nd is the Map of the `oldExchange` headers, and the 3rd is paired to the body of the `newExchange`, and the 4th parameter is the Map of the `newExchange` headers:

And finally if we have 6 parameters the we also have the properties of the Exchanges:

To use this with the `Aggregate` EIP we can use a POJO with the aggregate logic as follows:

And then in the Camel route we create an instance of our bean, and then refer to the bean in the route using `bean` method from `org.apache.camel.util.toolbox.AggregationStrategies` as shown:

We can also provide the bean type directly:

And if the bean has only one method we do not need to specify the name of the method:

And the `append` method could be static:

If you are using XML DSL then we need to declare a `<bean>` with the POJO:

And in the Camel route we use `strategyRef` to refer to the bean by its id, and the `strategyMethodName` can be used to define the method name to call:

When using XML DSL you must define the POJO as a `<bean>`.

Aggregating when no data

By default when using POJOs as `AggregationStrategy`, then the method is **only** invoked when there is data to be aggregated (by default). You can use the option `strategyMethodAllowNull` to configure this. Where as without using POJOs then you

may have `null` as `oldExchange` or `newExchange` parameters. For example the Aggregate EIP will invoke the `AggregationStrategy` with `oldExchange` as `null`, for the first Exchange incoming to the aggregator. And then for subsequent Exchanges then `oldExchange` and `newExchange` parameters are both not `null`.

Example with Content Enricher and no data

Though with POJOs as `AggregationStrategy` we made this simpler and only call the method when `oldExchange` and `newExchange` is not `null`, as that would be the most common use-case. If you need to allow `oldExchange` or `newExchange` to be `null`, then you can configure this with the POJO using the `AggregationStrategyBeanAdapter` as shown below. On the bean adapter we call `setAllowNullNewExchange` to allow the new exchange to be `null`.

This can be configured a bit easier using the `beanAllowNull` method from `AggregationStrategies` as shown:

Then the `append` method in the POJO would need to deal with the situation that `newExchange` can be `null`:

In the example above we use the Content Enricher EIP using `pollEnrich`. The `newExchange` will be `null` in the situation we could not get any data from the "seda:foo" endpoint, and therefore the timeout was hit after 1 second. So if we need to do some special merge logic we would need to set `setAllowNullNewExchange=true`, so the `append` method will be invoked. If we do not do that then when the timeout was hit, then the `append` method would normally not be invoked, meaning the Content Enricher did not merge/change the message.

In XML DSL you would configure the `strategyMethodAllowNull` option and set it to `true` as shown below:

Different body types

When for example using `strategyMethodAllowNull` as `true`, then the parameter types of the message bodies does not have to be the same. For example suppose we want to aggregate from a `com.foo.User` type to a `List<String>` that contains the user name. We could code a POJO doing this as follows:

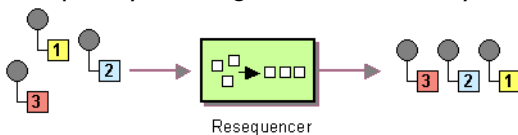
Notice that the return type is a `List` which we want to contain the user names. The 1st parameter is the list of names, and then notice the 2nd parameter is the incoming `com.foo.User` type.

See also

- The Loan Broker Example which uses an aggregator
- Blog post by Torsten Mielke about using the aggregator correctly.
- The old Aggregator
- HawtDB, LevelDB or SQL Component for persistence support
- Aggregate Example for an example application

Resequencer

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



Camel supports two resequencing algorithms:

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

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.

Using the Fluent Builders

This is equivalent to

The batch-processing resequencer can be further configured via the `size()` and `timeout()` methods.

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.



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.

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

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

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 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`.

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.

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.

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

The stream-processing resequencer can be further configured via the `capacity()` and `timeout()` methods.

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.

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.

```


```

or custom comparator via the `comparator()` method

```


```

or via a `StreamResequencerConfig` object.

```


```

Using the Spring XML Extensions

```


```

Further Examples

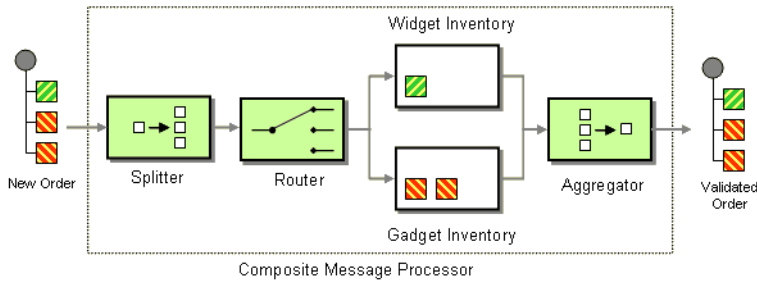
For further examples of this pattern in use you could look at the batch-processing resequencer junit test case and the stream-processing resequencer junit test case

Using This Pattern

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

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.

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.

Using the Spring XML Extensions

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.

The bean with the methods to transform the order line and process the order as well:



Using the splitter alone is often easier and possibly a better solution. So take a look at this first, before involving the aggregator.



Using XML

If you use XML, then the `<split>` tag offers the `strategyRef` attribute to refer to your custom `AggregationStrategy`

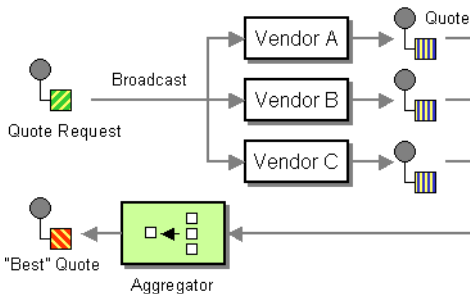
And the `AggregationStrategy` we use with the Splitter eip to combine the orders back again (eg fork/join):

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:

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

This message will be distributed to the following Endpoints: `bean:vendor1`, `bean:vendor2`, and `bean:vendor3`. These are all beans which look like

and are loaded up in Spring like

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

Finally, we expect to get the lowest quote of \$1 out of \$1, \$2, and \$3.

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

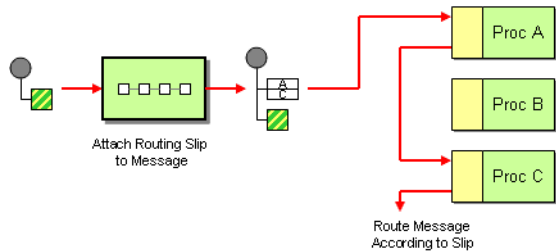
A full example of the static Scatter-Gather configuration can be found in the Loan Broker Example.

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

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.

Example

The following route will take any messages sent to the Apache ActiveMQ queue **SomeQueue** and pass them into the Routing Slip pattern.

Messages will be checked for the existance of the "aRoutingSlipHeader" header. The value of this header should be a comma-delimited list of endpoint URIs 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 though 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

Using the Spring XML Extensions

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.

And in Spring XML its an attribute on the recipient list tag.

Then lets say the `myHeader` contains the following two endpoints `direct:foo,xxx:bar`. The first endpoint is valid and works. However the 2nd is invalid and will just be ignored. Camel logs at INFO level, so you can see why the endpoint was invalid.

Expression supporting

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.

And in Spring XML its an attribute on the recipient list tag.

Further Examples

For further examples of this pattern in use you could look at the routing slip test cases.

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.

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

Name	Default Value	Description
maximumRequestsPerPeriod	Ê	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.
timePeriodMillis	1000	The time period in milliseconds, in which the throttler will allow at most <code>maximumRequestsPerPeriod</code> number of messages.
asyncDelayed	false	Camel 2.4: If enabled then any messages which is delayed happens asynchronously using a scheduled thread pool.
executorServiceRef	Ê	Camel 2.4: Refers to a custom Thread Pool to be used if <code>asyncDelay</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.

Examples

Using the Fluent Builders

So the above example will throttle messages all messages received on **sedas:a** before being sent to **mock:result** ensuring that a maximum of 3 messages are sent in any 10 second window.

Note that since `timePeriodMillis` 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...

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

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:

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:

```
.....
```

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.

```
.....
```

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.

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

Name	Default Value	Description
messageFrequency	Ê	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:

Using the Spring XML Extensions

And the same example in Spring XML is:

And since it uses a default of 1 second you can omit this configuration in case you also want to use 1 second

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.

See Also

- Throttler
- Aggregator

Delayer

The Delayer Pattern allows you to delay the delivery of messages to some destination.

Options

Name	Default Value	Description
asyncDelayed	false	Camel 2.4: If enabled then delayed messages happens asynchronously using a scheduled thread pool.
executorServiceRef	Ê	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

So the above example will delay all messages received on **seda:b** 1 second before sending them to **mock:result**.

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



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>

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

Spring DSL

The sample below demonstrates the delay in Spring DSL:

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 Spring XML

You use the `asyncDelayed="true"` attribute to enable the async behavior.

Creating a custom delay

You can use an expression to determine when to send a message using something like this

then the bean would look like this...

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.

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 jsessionid in the web or JMSXGroupID 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.

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.

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 `@deprecated ref` attribute.

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

Using the Spring configuration

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**.

Failover offers the following options:

Option	Type	Default	Description
--------	------	---------	-------------



Load balancing HTTP endpoints

If you are proxying and load balancing HTTP, then see this page for more details.



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.

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.

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.

Here is a sample to failover only if a `IOException` related exception was thrown:

```
failover:
  failoverOnException: true
```

You can specify multiple exceptions to failover as the option is varargs, for instance:

```
failover:
  failoverOnException: true, IOException, FileNotFoundException
```

Using failover in Spring DSL

Failover can also be used from Spring DSL and you configure it as:

```
failover:
  failoverOnException: true
```

Using failover in round robin mode

An example using Java DSL:

```
failover:
  failoverOnException: true
```

And the same example using Spring XML:

```
<failover>
  failoverOnException="true"
</failover>
```

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



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.



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.

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.

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 <code>distributionRatio</code> is a list consisting on integer weights passed in as a parameter. The <code>distributionRatio</code> 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



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.

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:

```
loadBalancer().roundRobin().distributionRatio("2,3,5").distributionRatioDelimiter(",");
```

And the same example using Spring XML:

```
<loadBalancer roundRobin="true" distributionRatio="2,3,5" distributionRatioDelimiter=","/>
```

Available In Camel 2.6

An example using Java DSL:

```
loadBalancer().roundRobin().distributionRatio("2,3,5").distributionRatioDelimiter(",");
```

And the same example using Spring XML:

```
<loadBalancer roundRobin="true" distributionRatio="2,3,5" distributionRatioDelimiter=","/>
```

Custom Load Balancer

You can use a custom load balancer (eg your own implementation) also.

An example using Java DSL:

And the same example using XML DSL:

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:

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:

Listing 1. Custom load balancer implementation

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.

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

Name	Default Value	Description
<code>strategyRef</code>	<code>Ê</code>	Refers to an <code>AggregationStrategy</code> 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 <code>AggregationStrategy</code> , see the Aggregate page for more details.
<code>strategyMethodName</code>	<code>Ê</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>parallelProcessing</code>	<code>false</code>	If enables 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.
<code>executorServiceRef</code>	<code>Ê</code>	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.
<code>stopOnException</code>	<code>false</code>	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 <code>AggregationStrategy</code> 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	Ê	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 <code>TimeoutAwareAggregationStrategy</code> then the <code>timeout</code> 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	Ê	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

By default Multicast invokes each endpoint sequentially. If parallel processing is desired, simply use

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:

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:

And using XML DSL you specify it as follows:

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 multicast.
For example if you have a mutable message body as this Animal class:

Listing 1. Animal

Then we can create a deep clone processor which clones the message body:

Listing 1. AnimalDeepClonePrepare

Then we can use the AnimalDeepClonePrepare class in the Multicast route using the `onPrepare` option as shown:

Listing 1. Multicast using onPrepare

And the same example in XML DSL

Listing 1. Multicast using onPrepare

Notice the `onPrepare` option is also available on other EIPs such as Splitter, Recipient List, and 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.

LOOP

The Loop allows for processing a message a number of times, possibly in a different way for each iteration. Useful mostly during testing.

Options

Name	Default Value	Description



Design for immutable

Its best practice to design for immutable objects.



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.

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.

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
<code>CamelLoopSize</code>	Total number of loops
<code>CamelLoopIndex</code>	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

Use expression to determine loop count

Use expression to determine loop count

Using the Spring XML Extensions

Pass loop count as an argument

Use expression to determine loop count

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.

However if we do **not** enable copy mode then "mock:loop" will receive "AB", "ABB", "ABBB", etc. messages.

The equivalent example in XML DSL in copy mode is as follows:

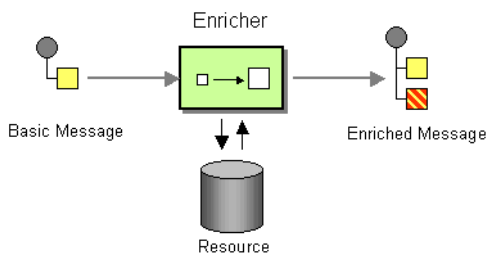
Using This Pattern

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

MESSAGE TRANSFORMATION

Content Enricher

Camel supports the Content Enricher from the EIP patterns using a Message Translator, 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)

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:

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

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

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

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

- TransformTest
- TransformViaDSLTest

Using Spring XML

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.

Enrich Options

Name	Default Value	Description
<code>uri</code>	<code>Ê</code>	The endpoint uri for the external service to enrich from. You must use either <code>uri</code> or <code>ref</code> .
<code>ref</code>	<code>Ê</code>	Refers to the endpoint for the external service to enrich from. You must use either <code>uri</code> or <code>ref</code> .
<code>strategyRef</code>	<code>Ê</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 <code>POJO</code> as the <code>AggregationStrategy</code> , see the Aggregate page for more details.
<code>strategyMethodName</code>	<code>Ê</code>	Camel 2.12: This option can be used to explicit declare the method name to use, when using <code>POJOs</code> as the <code>AggregationStrategy</code> . 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.

Using the Fluent Builders

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:

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

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.

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 just omit the `strategyRef` attribute:

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

Name	Default Value	Description
<code>uri</code>	<code>É</code>	The endpoint uri for the external service to enrich from. You must use either <code>uri</code> or <code>ref</code> .
<code>ref</code>	<code>É</code>	Refers to the endpoint for the external service to enrich from. You must use either <code>uri</code> or <code>ref</code> .

strategyRef	Ê	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.
strategyMethodName	Ê	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.
strategyMethodAllowNull	false	Camel 2.12: If this option is <code>false</code> then the <code>aggregate</code> 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.
timeout	-1	Timeout in millis when polling from the external service. See below for important details about the timeout.

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.

Example

In this example we enrich the message by loading the content from the file named `inbox/data.txt`.

And in XML DSL you do:

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:

Using This Pattern

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

Content Filter

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

- Message Translator
- invoking a Java bean
- Processor object



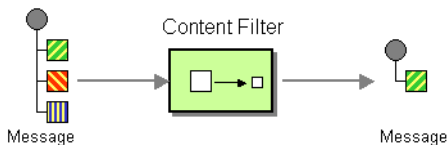
Good practice to use timeout value

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.



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.



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

In this example we add our own Processor

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

- TransformTest
- TransformViaDSLTest

Using Spring XML

You can also use XPath to filter out part of the message you are interested in:

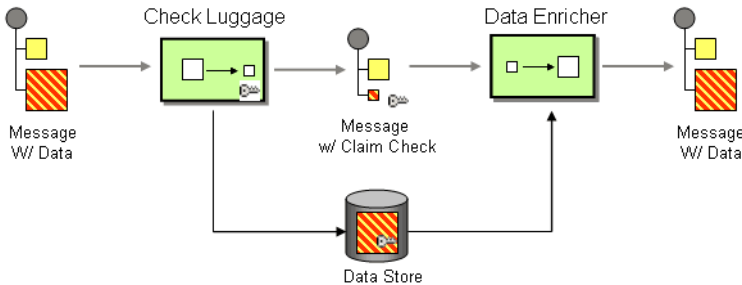
Using This Pattern

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

Using the Spring XML Extensions

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

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

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:

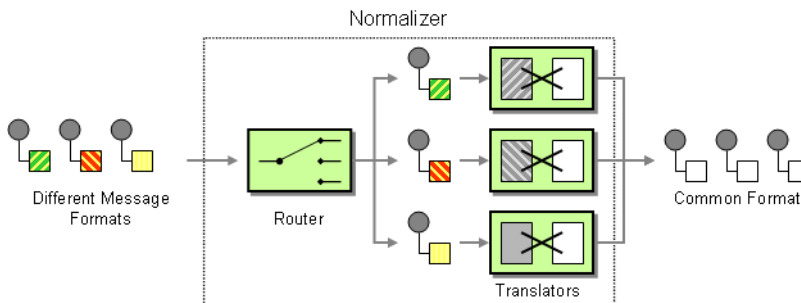
`camel-core/src/test/java/org/apache/camel/processor/ClaimCheckTest.java`

Using This Pattern

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

Normalizer

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



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

In this case we're using a Java bean as the normalizer. The class looks like this

Using the Spring XML Extensions

The same example in the Spring DSL

See Also

- Message Router

- Content Based Router
- Message Translator

Using This Pattern

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

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>	<code>ê</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.

```

route() {
    from("file://...")
    .tokenize(body(), "\n")
    .sort()
    .to("file://...")
}

```

You can pass in your own comparator as a 2nd argument:

```

route() {
    from("file://...")
    .tokenize(body(), "\n")
    .sort(myComparator())
    .to("file://...")
}

```

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://..." />
    <tokenize body "${body}" delimiter="\n" />
    <sort />
    <to uri="file://..." />
</route>

```

```

<!-- Listing 1. Camel 2.6 or older -->
<route>
    <from uri="file://..." />
    <tokenize body "${body}" delimiter="\n" />
    <sort comparator="myComparator" />
    <to uri="file://..." />
</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://..." />
    <tokenize body "${body}" delimiter="\n" />
    <sort ref="myComparator" />
    <to uri="file://..." />
</route>

```

Listing 1. Camel 2.6 or older

Besides `<simple>`, you can supply an expression using any language you like, so long as it returns a list.

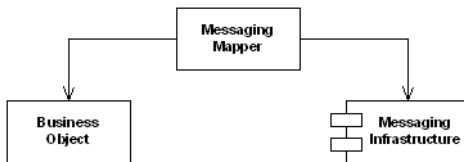
Using This Pattern

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.



See also

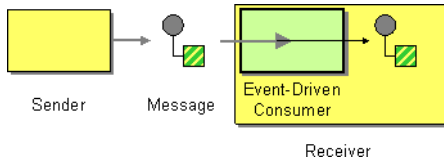
- Message Translator
- Type Converter
- CXF for JAX-WS support for binding business logic to messaging & web services
- Pojo
- Bean

Using This Pattern

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

Event Driven Consumer

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



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

For more details see

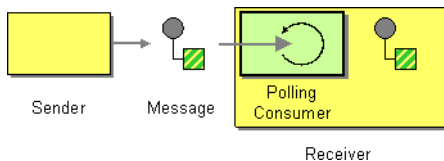
- Message
- Message Endpoint

Using This Pattern

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

Polling Consumer

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



So in your Java code you can do

The ConsumerTemplate (discussed below) is also available.

There are 3 main polling methods on PollingConsumer

Method name	Description
receive()	Waits until a message is available and then returns it; potentially blocking forever

<code>receive(long)</code>	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
<code>receiveNoWait()</code>	Attempts to receive a message exchange immediately without waiting and returning null if a message exchange is not available yet

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 3 operations above, but also including convenient methods for returning the body, etc `consumeBody`.

The example from above using `ConsumerTemplate` is:

Or to extract and get the body you can do:

And you can provide the body type as a parameter and have it returned as the type:

You get hold of a `ConsumerTemplate` from the `CamelContext` with the `createConsumerTemplate` operation:

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:

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.

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:

And then we have out logic in our bean:

Scheduled Poll Components

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

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

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

For more details see:

- PollingConsumer
- Scheduled Polling Components
 - ScheduledPollConsumer
 - Atom
 - File
 - FTP
 - hbase
 - iBATIS
 - JPA
 - Mail
 - MyBatis
 - Quartz
 - SNMP
 - AWS-S3
 - AWS-SQS

ScheduledPollConsumer Options

The ScheduledPollConsumer supports the following options:

Option	Default	Description
pollStrategy	.	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 have been created and being 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 <code>WARN</code> level and ignore it.
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.
initialDelay	1000	Milliseconds before the first poll starts.
delay	500	Milliseconds before the next poll.
useFixedDelay	£	Controls if fixed delay or fixed rate is used. See <code>ScheduledExecutorService</code> in JDK for details. In Camel 2.7.x or older the default value is <code>false</code> . From Camel 2.8 onwards the default value is <code>true</code> .

timeUnit	TimeUnit.MILLISECONDS	time unit for initialDelay and delay options.
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.
greedy	false	Camel 2.10.6/2.11.1: If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.
scheduler	null	Camel 2.12: Allow to plugin a custom <code>org.apache.camel.spi.ScheduledPollConsumerScheduler</code> to use as the scheduler for firing when the polling consumer runs. The default implementation uses the <code>ScheduledExecutorService</code> and there is a Quartz2, and Spring based which supports CRON expressions. Notice: If using a custom scheduler then the options for <code>initialDelay</code> , <code>useFixedDelay</code> , <code>timeUnit</code> , and <code>scheduledExecutorService</code> may not be in use. Use the text <code>quartz2</code> to refer to use the Quartz2 scheduler; and use the text <code>spring</code> to use the Spring based; and use the text <code>#myScheduler</code> 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.
backoffMultiplier	0	Camel 2.12: To let the scheduled polling consumer backoff 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.
backoffIdleThreshold	0	Camel 2.12: The number of subsequent idle polls that should happen before the <code>backoffMultiplier</code> should kick-in.
backoffErrorThreshold	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.

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 onwards 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 backoff if its becoming idle for a while you can do:

```

-----
In this example, the FTP consumer will poll for new FTP files evert 5th second. But if it has
been idle for 5 attempts in a row, then it will
backoff using a multiplier of 6, which means it will now poll every 5 x 6 = 30th second instead.
When the consumer eventually pickup a file, then the backoff 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 backoff 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`.

Controlling the error handling 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 3 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** onwards 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** onwards 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 3 times:

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`:

Notice: 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`:

Using This Pattern

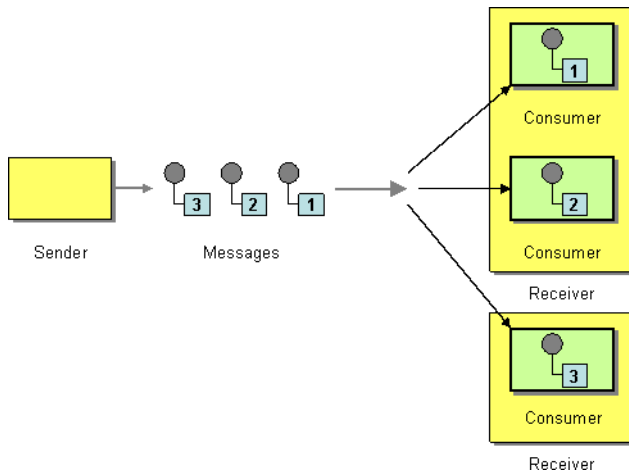
If you would like to use this EIP Pattern then please read the *Getting Started*, you may also find the *Architecture* useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

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

```

<code>
</code>

```

or in Spring DSL

```

<code>
</code>

```

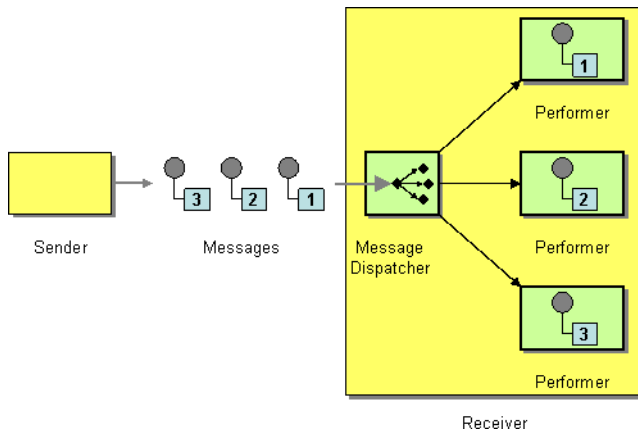
Or just run multiple JVMs of any ActiveMQ or JMS route 😊

Using This Pattern

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

Message Dispatcher

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



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

See Also

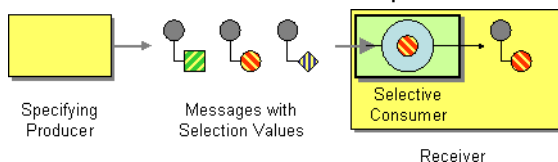
- JMS
- Selective Consumer
- Content Based Router
- Endpoint

Using This Pattern

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

Selective Consumer

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



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

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

Using the Fluent Builders

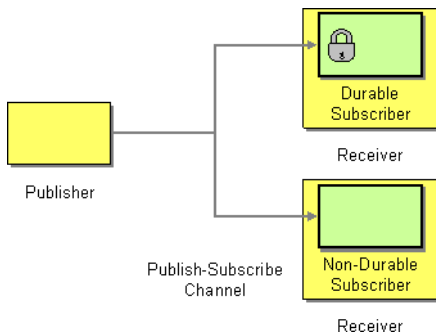
Using the Spring XML Extensions

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

Using the Spring XML Extensions

Here is another example of JMS durable subscribers, but this time using virtual topics (recommended by AMQ over durable subscriptions)

Using the Fluent Builders

Using the Spring XML Extensions

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

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.
messageldRepositoryRef	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.

Using the Fluent Builders

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

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

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

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.

Listing 1. Filter duplicate messages

The sample example in XML DSL would be:

Listing 1. Filter duplicate messages

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:

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.

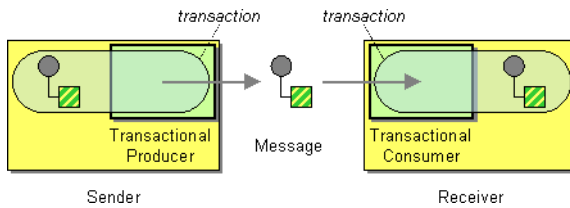
See this little tutorial, how setup such an idempotent repository on two cluster nodes using Apache Karaf.

Using This Pattern

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

Transactional Client

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



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

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

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

You first define needed object in the spring configuration.

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

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:

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

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

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.

And then refer to "required" from the route:



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.

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.

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.

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.

And then we configure our routes. Notice that all we have to do is mark the route as transacted using the **transacted** tag.



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.

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_REQUIRES_NEW`

This is configured in the Spring XML file:

```
<!-- ... -->
```

Then in the routes you use `transacted` DSL to indicate which of these two propagations it uses.

```
<!-- ... -->
```

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

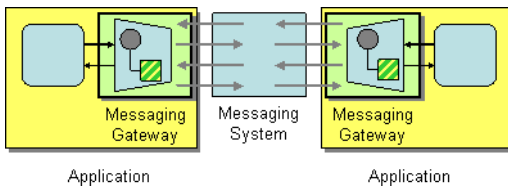
- Error handling in Camel
- `TransactionErrorHandler`
- Error Handler
- JMS

Using This Pattern

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

Messaging Gateway

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



Components like Bean 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.

See Also

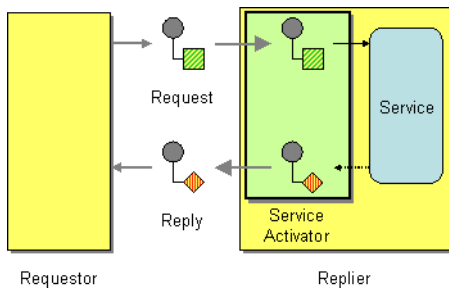
- Bean
- CXF
- Using CamelProxy
- POJO Producing
- Spring Remoting

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

Using the Spring XML Extensions

See Also

- Bean
- Pojo
- CXF

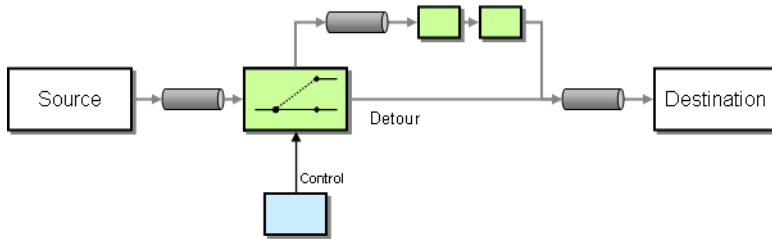
Using This Pattern

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..

Using the Spring XML Extensions

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:

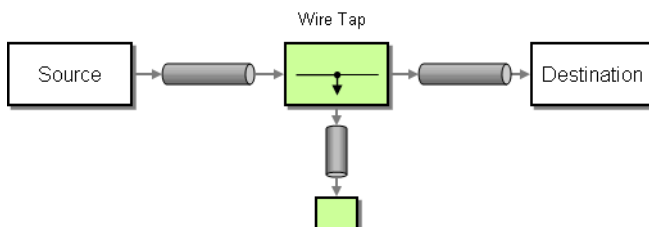
`camel-core/src/test/java/org/apache/camel/processor/DetourTest.java`

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 Value	Description
uri	Ê	The URI of the endpoint to which the wire-tapped message will be sent. You should use either <code>uri</code> or <code>ref</code> .
ref	Ê	Reference identifier of the endpoint to which the wire-tapped message will be sent. You should use either <code>uri</code> or <code>ref</code> .
executorServiceRef	Ê	Reference identifier of a custom Thread Pool to use when processing the wire-tapped messages. If not set, Camel will use a default thread pool.
processorRef	Ê	Reference identifier of a custom Processor to use for creating a new message (e.g., the "send a new message" mode). See below.
copy	true	Camel 2.3: Whether to copy the Exchange before wire-tapping the message.
onPrepareRef	Ê	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.

WireTap thread pool

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.
- 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

```


```

Using the Spring XML Extensions

```


```

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, et cetera. An Expression can only be used to set the IN body.

From **Camel 2.3** onwards 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 in `false` to create a new, empty Exchange.

Here is the Expression variation. This example is from Camel 2.3, where we disable `copy` by passing in `false` to create a new, empty Exchange.

Using the Spring XML Extensions

The processor variation, which uses a **processorRef** attribute to refer to a Spring bean by ID:

Here is the Expression variation, where the expression is defined in the **body** tag:

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"

Further Example

For another example of this pattern, refer to the wire tap test case.

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. In Camel 2.8 onwards, you can now set headers as well in 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

XML DSL

The XML DSL is slightly different than Java DSL in how you configure the message body and headers using `<body>` and `<setHeader>`:

Using `onPrepare` to execute custom logic when preparing messages

Available as of Camel 2.8

See details at [Multicast](#)

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.

LOG

How can I log processing a Message?

Camel provides many ways to log processing a message. Here is just some 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

And 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

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.

The `log` DSL have overloaded methods to set the logging level and/or name as well.



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.



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.

For example you can use this to log the file name being processed if you consume files.

Using log DSL from Spring

In Spring DSL its also easy to use log DSL as shown below:

The log tag has attributes to set the `message`, `loggingLevel` and `logName`. For example:

Using slf4j Marker

Available as of Camel 2.9

You can specify a marker name in the DSL

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.

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`.

URI format

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:

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

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

Options

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



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.

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.

Configuring the Connection Factory using Spring XML

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

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:

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

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 `maxActive` you can use to configure the maximum number of sessions per connection; the default value is 500. From **ActiveMQ 5.7** onwards the option has been renamed to better reflect its purpose, being named as `maxActiveSessionPerConnection`. 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.



Notice the **init** and **destroy** methods on the pooled connection factory. This is important to ensure the connection pool is properly started and shutdown.

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

Then use it in your Camel route as follows

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:

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

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
and containing string:

Getting Component JAR

You will need this dependency

- `activemq-camel`

ActiveMQ is an extension of the JMS component released with the ActiveMQ project.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

ACTIVEMQ JOURNAL COMPONENT

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

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

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

URI format

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

Options

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

You can append query options to the URI in the following format,

`?option=value&option=value&...`

Expected Exchange Data Types

The consumer of a Journal endpoint generates DefaultExchange objects with the in message :

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

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

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

AMQP

Available as of Camel X.X

The **amqp** component supports the AMQP protocol using the Client API of the Qpid project.

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

```
<code><pre></pre></code>
```

URI format

```
<code><pre></pre></code>
```

AMQP Options

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

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

SQS COMPONENT

Available as of Camel 2.6

The sqs component supports sending and receiving messages to Amazon's SQS service.

URI Format

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
amazonSQSClient	null	Shared	Reference to a <code>com.amazonaws.services.sqs.AmazonSQS</code> in the Registry.
accessKey	null	Shared	Amazon AWS Access Key
secretKey	null	Shared	Amazon AWS Secret Key
amazonSQSEndpoint	null	Shared	The region with which the AWS-SQS client wants to work with.
attributeNames	null	Consumer	A list of attributes to set in the <code>com.amazonaws.services.sqs.model.ReceiveMessageRequest</code> .
defaultVisibilityTimeout	null	Shared	The visibility timeout (in seconds) to set in the <code>com.amazonaws.services.sqs.model.CreateQueueRequest</code> .
deleteAfterRead	true	Consumer	Delete message from SQS after it has been read
maxMessagesPerPoll	null	Consumer	The maximum number of messages which can be received in one poll to set in the <code>com.amazonaws.services.sqs.model.ReceiveMessageRequest</code> .
visibilityTimeout	null	Shared	The duration (in seconds) that the received messages are hidden from subsequent retrieve requests after being retrieved by a <code>ReceiveMessage</code> request to set in the <code>com.amazonaws.services.sqs.model.SetQueueAttributesRequest</code> . This only make sense if its different from <code>defaultVisibilityTimeout</code> . It changes the queue visibility timeout attribute permanently. Camel 2.8: The duration (in seconds) that the received messages are hidden from subsequent retrieve requests after being retrieved by a <code>ReceiveMessage</code> request to set in the <code>com.amazonaws.services.sqs.model.ReceiveMessageRequest</code> . It does NOT change the queue visibility timeout attribute permanently.
messageVisibilityTimeout	null	Consumer	Camel 2.10: If enabled then a scheduled background task will keep extending the message visibility on SQS. This is needed if it takes a long time to process the message. If set to true <code>defaultVisibilityTimeout</code> must be set. See details at Amazon docs.
extendMessageVisibility	false	Consumer	Camel 2.8: The <code>maximumMessageSize</code> (in bytes) an SQS message can contain for this queue, to set in the <code>com.amazonaws.services.sqs.model.SetQueueAttributesRequest</code> .
maximumMessageSize	null	Shared	Camel 2.8: The <code>messageRetentionPeriod</code> (in seconds) a message will be retained by SQS for this queue, to set in the <code>com.amazonaws.services.sqs.model.SetQueueAttributesRequest</code> .
messageRetentionPeriod	null	Shared	Camel 2.8: The policy for this queue to set in the <code>com.amazonaws.services.sqs.model.SetQueueAttributesRequest</code> .
policy	null	Shared	Camel 2.9.3: Delay sending messages for a number of seconds.
delaySeconds	null	Producer	Camel 2.11: Duration in seconds (0 to 20) that the <code>ReceiveMessage</code> action call will wait until a message is in the queue to include in the response.
waitTimeSeconds	0	Producer	Camel 2.11: If you do not specify <code>WaitTimeSeconds</code> in the request, the queue attribute <code>ReceiveMessageWaitTimeSeconds</code> is used to determine how long to wait.
receiveMessageWaitTimeSeconds	0	Shared	Camel 2.12: Specify the queue owner aws account id when you need to connect the queue with different account owner.
queueOwnerAWSAccountId	null		



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](#).



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
<code>CamelAwsSqsMD5OfBody</code>	<code>String</code>	The MD5 checksum of the Amazon SQS message.
<code>CamelAwsSqsMessageId</code>	<code>String</code>	The Amazon SQS message ID.
<code>CamelAwsSqsDelaySeconds</code>	<code>Integer</code>	Since Camel 2.11 , the delay seconds that the Amazon SQS message can be see by others.

Message headers set by the SQS consumer

Header	Type	Description
<code>CamelAwsSqsMD5OfBody</code>	<code>String</code>	The MD5 checksum of the Amazon SQS message.
<code>CamelAwsSqsMessageId</code>	<code>String</code>	The Amazon SQS message ID.
<code>CamelAwsSqsReceiptHandle</code>	<code>String</code>	The Amazon SQS message receipt handle.
<code>CamelAwsSqsAttributes</code>	<code>Map<String, String></code>	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:

```
-----
```

and refer to it in your Camel `aws-sqs` component configuration:

Dependencies

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

Listing 1. pom.xml

where `${camel-version}` must be replaced by the actual version of Camel (2.6 or higher).

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:

URI format

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

Options

Property	Default	Description
splitEntries	true	If <code>true</code> 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 <code>false</code> then Camel will poll a fresh feed on every invocation.
filter	true	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.
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 <code>true</code> , only one entry is processed per <code>consumer.delay</code> . Only applicable when <code>splitEntries</code> is set to <code>true</code> .
feedHeader	true	Sets whether to add the Abdera Feed object as a header.
sortEntries	false	If <code>splitEntries</code> is <code>true</code> , this sets whether to sort those entries by updated date.
consumer.delay	60000	Delay in millis between each poll.

```
consumer.initialDelay 1000 Millis before polling starts.
consumer.userFixedDelay false If true, use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
```

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>`.

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

Camel can set the `Feed` object on the `In` header (see `feedHeader` option to disable this):

Message Headers

Camel atom uses these headers.

Header	Description
<code>CamelAtomFeed</code>	When consuming the <code>org.apache.abdera.model.Feed</code> object is set to this header.

Samples

In this sample we poll James Strachan's blog.

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.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [RSS](#)

BEAN COMPONENT

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

URI format

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

Options

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 <code>true</code> , the In message body should be an array of parameters.

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

Using

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.

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

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:

And the same route using Spring DSL:

Bean as endpoint

Camel also supports invoking Bean as an Endpoint. In the route below:

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:

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:

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:

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.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [Class component](#)
- [Bean Binding](#)
- [Bean Integration](#)

BEAN VALIDATION COMPONENT

Available as of Camel 2.3

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

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

URI format

or

Where **something** must be present to provide a valid url

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

URI Options

Option	Default	Description
group	javax.validation.groups.Default	The custom validation group to use.
messageInterpolator	org.hibernate.validator.engine.ResourceBundleMessageInterpolator	Reference to a custom javax.validation.MessageInterpolator in the Registry.
traversableResolver	org.hibernate.validator.engine.resolver.DefaultTraversableResolver	Reference to a custom javax.validation.TraversableResolver in the Registry.
constraintValidatorFactory	org.hibernate.validator.engine.ConstraintValidatorFactoryImpl	Reference to a custom javax.validation.ConstraintValidatorFactory in the Registry.

ServiceMix4/OSGi Deployment.

The bean-validator when deployed in an OSGi environment requires a little help to accommodate the resource loading specified in JSR303, this was fixed in Servicemix-Specs 1.6-SNAPSHOT.

Example

Assumed we have a java bean with the following annotations

```
Listing 1. Car.java
```

and an interface definition for our custom validation group

```
Listing 1. OptionalChecks.java
```

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

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

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

```
Listing 1. AllChecks.java
```

and then your route definition should looks like this

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

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

```
Listing 1. validation.xml
```

and the constraints-car.xml file

Listing 1. constraints-car.xml

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

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

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:

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

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

CACHE COMPONENT

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:

```
<code></code>
```

URI format

```
<code></code>
```

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 defined cache
<code>memoryStoreEvictionPolicy</code>	<code>MemoryStoreEvictionPolicy.LFU</code>	The number of elements that may be stored in the defined cache. Options include <ul style="list-style-type: none"> <code>MemoryStoreEvictionPolicy.LFU</code> - Least frequently used <code>MemoryStoreEvictionPolicy.LRU</code> - Least recently used <code>MemoryStoreEvictionPolicy.FIFO</code> - first in first out, the oldest element by creation time
<code>overflowToDisk</code>	true	Specifies whether cache may overflow to disk
<code>eternal</code>	false	Sets whether elements are eternal. If eternal, timeouts are ignored and the element never expires.
<code>timeToLiveSeconds</code>	300	The maximum time between creation time and when an element expires. Is used only if the element is not eternal
<code>timeToIdleSeconds</code>	300	The maximum amount of time between accesses before an element expires
<code>diskPersistent</code>	false	Whether the disk store persists between restarts of the Virtual Machine.
<code>diskExpiryThreadIntervalSeconds</code>	120	The number of seconds between runs of the disk expiry thread.
<code>cacheManagerFactory</code>	null	<p>Camel 2.8: If you want to use a custom factory which instantiates and creates the <code>EHCache net.sf.ehcache.CacheManager</code>.</p> <p>Type: <code>abstract org.apache.camel.component.cache.CacheManagerFactory</code></p>
<code>eventListenerRegistry</code>	null	<p>Camel 2.8: Sets a list of <code>EHCache net.sf.ehcache.event.CacheEventListener</code> for all new caches- no need to define it per cache in <code>EHCache.xml</code> config anymore.</p> <p>Type: <code>org.apache.camel.component.cache.CacheEventListenerRegistry</code></p>
<code>cacheLoaderRegistry</code>	null	<p>Camel 2.8: Sets a list of <code>org.apache.camel.component.cache.CacheLoaderWrapper</code> that extends <code>EHCache net.sf.ehcache.loader.CacheLoader</code> for all new caches- no need to define it per cache in <code>EHCache.xml</code> config anymore.</p> <p>Type: <code>org.apache.camel.component.cache.CacheLoaderRegistry</code></p>
<code>key</code>	null	Camel 2.10: To configure using a cache key by default. If a key is provided in the message header, then the key from the header takes precedence.
<code>operation</code>	null	Camel 2.10: To configure using a cache operation by default. If an operation in the message header, then the operation from the header takes precedence.

Sending/Receiving Messages to/from the cache

Message Headers up to Camel 2.7

Header	Description
	The operation to be performed on the cache. Valid options are <ul style="list-style-type: none">• GET• CHECK• ADD• UPDATE• DELETE• DELETEALL GET and CHECK requires Camel 2.3 onwards.
CACHE_OPERATION	
CACHE_KEY	The cache key used to store the Message in the cache. The cache key is optional if the CACHE_OPERATION is DELETEALL

Message Headers Camel 2.8+

Header	Description
	The operation to be performed on the cache. The valid options are <ul style="list-style-type: none">• CamelCacheGet• CamelCacheCheck• CamelCacheAdd• CamelCacheUpdate• CamelCacheDelete• CamelCacheDeleteAll
CamelCacheOperation	
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.



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.

- 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

Example 2: Adding keys to the cache

Example 2: Updating existing keys in a cache

Example 3: Deleting existing keys in a cache

Example 4: Deleting all existing keys in a cache

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

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

Example 7: Getting an entry from the Cache

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.

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:

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

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.

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:name?method=method&multiParameterArray=true
```

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.
multiParameterArray	boolean	false	How to treat the parameters which are passed from the message body; if it is <code>true</code> , the In message body should be an array of parameters.

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

Using

You simply use the **class** component just as the Bean component but by specifying the fully qualified classname instead.

For example to use the `MyFooBean` you have to do as follows:

```
class:MyFooBean?method=hello
```

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



It might be useful to read the EHCACHE manual to get a better understanding of the Camel Cache replication mechanism.

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:

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

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 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:

URI format

The **channelName** represents a topic that can be subscribed to by the Camel endpoints.



See more

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

Examples

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

where `cometds`: represents an SSL configured endpoint.

See this blog entry by David Greco who contributed this component to Apache Camel, for a full sample.

Options

Name	Default Value	Description
<code>resourceBase</code>	<code>É</code>	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 <code>baseResource</code> from Camel 2.7 onwards.
<code>baseResource</code>	<code>É</code>	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. Classpath is required for OSGI deployment where the resources are packaged in the jar
<code>timeout</code>	<code>240000</code>	The server side poll timeout in milliseconds. This is how long the server will hold a reconnect request before responding.
<code>interval</code>	<code>0</code>	The client side poll timeout in milliseconds. How long a client will wait between reconnects
<code>maxInterval</code>	<code>30000</code>	The max client side poll timeout in milliseconds. A client will be removed if a connection is not received in this time.
<code>multiFrameInterval</code>	<code>1500</code>	The client side poll timeout, if multiple connections are detected from the same browser.
<code>jsonCommented</code>	<code>true</code>	If <code>true</code> , the server will accept JSON wrapped in a comment and will generate JSON wrapped in a comment. This is a defence against Ajax Hijacking.
<code>logLevel</code>	<code>1</code>	<code>0=none, 1=info, 2=debug</code> .
<code>sslContextParameters</code>	<code>É</code>	Camel 2.9: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry.É This reference overrides any configured <code>SSLContextParameters</code> at the component level.É See Using the JSSE Configuration Utility.
<code>crossOriginFilterOn</code>	<code>false</code>	Camel 2.10: If <code>true</code> , the server will support for cross-domain filtering
<code>allowedOrigins</code>	<code>*</code>	Camel 2.10: The origins domain that support to cross, if the <code>crossOriginFilterOn</code> is <code>true</code>
<code>filterPath</code>	<code>É</code>	Camel 2.10: The <code>filterPath</code> will be used by the <code>CrossOriginFilter</code> , if the <code>crossOriginFilterOn</code> is <code>true</code>
<code>disconnectLocalSession</code>	<code>true</code>	Camel 2.10.5/2.11.1: (Producer only): Whether to disconnect local sessions after publishing a message to its channel. Disconnecting local session is needed as they are not swept by default by CometD, and therefore you can run out of memory.

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.Ê This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels.Ê The following examples demonstrate how to use the utility with the Cometd component.

Programmatic configuration of the component

Spring DSL based configuration of endpoint

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

CONTEXT COMPONENT

Available as of Camel 2.7

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:

URI format

Or you can omit the "context:" prefix.

- **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 URIs **supplyChain:invoices** or **supplyChain:purchaseOrders** to omit the physical endpoint kind and use pure logical URIs.

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...

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

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...

Naming endpoints

A context component instance can have many public input and output endpoints that can be accessed from outside it's 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:

```
<code><pre></pre></code>
```

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

```
<code><pre></pre></code>
```

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

In order to correctly function, 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

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.
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 for 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 way to sign and verify an exchange is with a `KeyPair` as follows.

```

// ...

```

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

```

<!-- ...

```

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.

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

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.

or

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

or

5) Changing the buffersize

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

or

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"`

or

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

- `Exchange.KEYSTORE_ALIAS, "CamelSignatureKeyStoreAlias"`

or

The header would be set as follows

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [Crypto](#) Crypto is also available as a Data Format

CXF COMPONENT

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

- [CXF Component](#)
- [URI format](#)
- [Options](#)
- [The descriptions of the dataformats](#)
- [How to enable CXF's LoggingOutInterceptor in MESSAGE mode](#)
- [Description of relayHeaders option](#)
- [Available only in POJO mode](#)
- [Changes since Release 2.0](#)
- [Configure the CXF endpoints with Spring](#)
- [Configuring the CXF Endpoints with Apache Aries Blueprint.](#)
- [How to make the camel-cxf component use log4j instead of java.util.logging](#)
- [How to let camel-cxf response message with xml start document](#)
- [How to consume a message from a camel-cxf endpoint in POJO data format](#)



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.

- How to prepare the message for the camel-cxf endpoint in POJO data format
- How to deal with the message for a camel-cxf endpoint in PAYLOAD data format
- How to get and set SOAP headers in POJO mode
- How to get and set SOAP headers in PAYLOAD mode
- SOAP headers are not available in MESSAGE mode
- How to throw a SOAP Fault from Camel
- How to propagate a camel-cxf endpoint's request and response context
- Attachment Support
- Streaming Support in PAYLOAD mode
- See Also

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

URI format

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.

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:

Options

Name	Required	Description
<code>wsdlURL</code>	No	The location of the WSDL. It is obtained from endpoint address by default. <i>Example: file://local/wsdl/hello.wsdl or wsdl/hello.wsdl</i>



CXF dependencies

If you want to learn about CXF dependencies you can checkout the [WHICH-JARS](#) text file.

serviceClass	Yes	<p>The name of the SEI (Service Endpoint Interface) class. This class can have, but does not require, JSR181 annotations.</p> <p>This option is only required by POJO mode. If the wsdlURL option is provided, serviceClass is not required for PAYLOAD and MESSAGE mode. When wsdlURL option is used without serviceClass, the serviceName and portName (endpointName for Spring configuration) options MUST be provided. It is possible to use # notation to reference a serviceClass object instance from the registry. E.g. serviceClass=#beanName. The serviceClass for a CXF producer (that is, the to endpoint) should be a Java interface.</p> <p>Since 2.8, it is possible to omit both wsdlURL and serviceClass options for PAYLOAD and MESSAGE mode. When they are omitted, arbitrary XML elements can be put in CxfPayload's body in PAYLOAD mode to facilitate CXF Dispatch Mode.</p> <p>Please be advised that the referenced object cannot be a Proxy (Spring AOP Proxy is OK) as it relies on Object.getClass().getName() method for non Spring AOP Proxy.</p> <p><i>Example:</i> org.apache.camel.Hello</p>
serviceName	No	<p>The service name this service is implementing, it maps to the wsdl:serviceName.</p> <p>Required for camel-cxf consumer since camel-2.2.0 or if more than one serviceName is present in WSDL.</p> <p><i>Example:</i> {http://org.apache.camel}ServiceName</p> <p>The port name this service is implementing, it maps to the wsdl:portName.</p>
portName	No	<p>Required for camel-cxf consumer since camel-2.2.0 or if more than one portName is present under serviceName.</p> <p><i>Example:</i> {http://org.apache.camel}PortName</p>
dataFormat	No	<p>The data type messages supported by the CXF endpoint.</p> <p><i>Default:</i> POJO <i>Example:</i> POJO, PAYLOAD, MESSAGE</p>
relayHeaders	No	<p>Please see the Description of relayHeaders option section for this option. Should a CXF endpoint relay headers along the route. Currently only available when dataFormat=POJO</p> <p><i>Default:</i> true <i>Example:</i> true, false</p>
wrapped	No	<p>Which kind of operation that CXF endpoint producer will invoke</p> <p><i>Default:</i> false <i>Example:</i> true, false</p>
wrappedStyle	No	<p>New in 2.5.0 The WSDL style that describes how parameters are represented in the SOAP body. If the value is false, CXF will chose the document-literal unwrapped style, If the value is true, CXF will chose the document-literal wrapped style</p> <p><i>Default:</i> Null <i>Example:</i> true, false</p>
setDefaultBus	No	<p>Will set the default bus when CXF endpoint create a bus by itself</p> <p><i>Default:</i> false <i>Example:</i> true, false</p>
bus	No	<p>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.</p> <p><i>Example:</i> bus=#busName</p>
cxfBinding	No	<p>Use # notation to reference a CXF binding object from the registry. The referenced object must be an instance of org.apache.camel.component.cxf.CxfBinding (use an instance of org.apache.camel.component.cxf.DefaultCxfBinding).</p> <p><i>Example:</i> cxfBinding=#bindingName</p>

headerFilterStrategy	No	Use # notation to reference a header filter strategy object from the registry. The referenced object must be an instance of <code>org.apache.camel.spi.HeaderFilterStrategy</code> (use an instance of <code>org.apache.camel.component.cxf.CxfHeaderFilterStrategy</code>).
		<i>Example:</i> <code>headerFilterStrategy=#strategyName</code>
loggingFeatureEnabled	No	New in 2.3. This option enables CXF Logging Feature which writes inbound and outbound SOAP messages to log. <i>Default:</i> <code>false</code> <i>Example:</i> <code>loggingFeatureEnabled=true</code>
defaultOperationName	No	New in 2.4, this option will set the default operationName that will be used by the CxfProducer which invokes the remote service. <i>Default:</i> <code>null</code> <i>Example:</i> <code>defaultOperationName=greetMe</code>
defaultOperationNamespace	No	New in 2.4. This option will set the default operationNamespace that will be used by the CxfProducer which invokes the remote service. <i>Default:</i> <code>null</code> <i>Example:</i> <code>defaultOperationNamespace=http://apache.org/hello_world_soap_http</code>
synchronous	No	New in 2.5. This option will let cxf endpoint decide to use sync or async API to do the underlying work. The default value is false which means camel-cxf endpoint will try to use async API by default. <i>Default:</i> <code>false</code> <i>Example:</i> <code>synchronous=true</code>
publishedEndpointUrl	No	New in 2.5. This option can override the endpointUrl that published from the WSDL which can be accessed with service address url plus ?wsdl. <i>Default:</i> <code>null</code> <i>Example:</i> <code>publishedEndpointUrl=http://example.com/service</code>
properties.XXX	No	Camel 2.8: Allows to set custom properties to CXF in the endpoint uri. For example setting <code>properties.mtom-enabled=true</code> to enable MTOM.
allowStreaming	No	New in 2.8.2. This option controls whether the CXF component, when running in PAYLOAD mode (see below), will DOM parse the incoming messages into DOM Elements or keep the payload as a <code>javax.xml.transform.Source</code> object that would allow streaming in some cases.
skipFaultLogging	No	New in 2.11. This option controls whether the PhaseInterceptorChain skips logging the Fault that it catches.

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	PAYLOAD is the message payload (the contents of the <code>soap:body</code>) 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 interceptor 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 , <code>CXF_MESSAGE</code> 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, `CamelCXFDataFormat`. The exchange key constant is defined in `org.apache.camel.component.cxf.CxfConstants.DATA_FORMAT_PROPERTY`.

How to enable CXF's LoggingOutInterceptor in MESSAGE mode

CXF's `LoggingOutInterceptor` outputs outbound message that goes on the wire to logging system (Java Util Logging). Since the `LoggingOutInterceptor` is in `PRE_STREAM`

phase (but `PRE_STREAM` phase is removed in `MESSAGE` mode), you have to configure `LoggingOutInterceptor` to be run during the `WRITE` phase. The following is an example.

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.

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.

Then, your endpoint can reference the `CxfHeaderFilterStrategy`.

- 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:

- 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 <i>Type: boolean</i> <i>Default: true</i>
<code>relayAllMessageHeaders</code>	No	All message headers will be propagated (without processing by Message Header Filters) <i>Type: boolean</i> <i>Default: false</i>
<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 <i>Type: boolean</i> <i>Default: false</i>

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.

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 `<cx:cxEndpoint/>` tag--these are required because the combined `{namespace}localName` syntax is presently not supported for this tag's attribute values.

The `cx:cxEndpoint` 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
cx:inInterceptors	The incoming interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
cx:inFaultInterceptors	The incoming fault interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
cx:outInterceptors	The outgoing interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
cx:outFaultInterceptors	The outgoing fault interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
cx:properties	A properties map which should be supplied to the JAX-WS endpoint. See below.
cx:handlers	A JAX-WS handler list which should be supplied to the JAX-WS endpoint. See below.
cx:dataBinding	You can specify the which <code>DataBinding</code> will be use in the endpoint. This can be supplied using the Spring <code><bean class="MyDataBinding"/></code> syntax.
cx:binding	You can specify the <code>BindingFactory</code> for this endpoint to use. This can be supplied using the Spring <code><bean class="MyBindingFactory"/></code> syntax.
cx:features	The features that hold the interceptors for this endpoint. A list of <code>{{<bean>}}s</code> or <code>{{<ref>}}s</code>
cx:schemaLocations	The schema locations for endpoint to use. A list of <code>{{<schemaLocation>}}s</code>
cx:serviceFactory	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 which show how to provide interceptors , properties and handlers here:

<http://cwiki.apache.org/CXF20DOC/jax-ws-configuration.html>

NOTE

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

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

Currently the endpoint element is the first supported CXF namespacehandler.

You can also use the bean references just as in spring

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'>`"

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

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

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

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.

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 want to get the object array from the message body, you can get the body using `message.getBody(Object[].class)`, as follows:

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.

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:

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.

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).

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

Then throw it as you like

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.

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)`:

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:

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 since 2.1. So, it is possible to retrieve attachments by Camel Message API

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.)

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

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

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

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.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

CXF BEAN COMPONENT

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



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

URI format

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

Options

Name	Description	Example	Required?	Default
<code>bus</code>	CXF bus reference specified by the # notation. The referenced object must be an instance of <code>org.apache.cxf.Bus</code> .	<code>bus=#busName</code>	No	Default bus Factory
<code>cxfBeanBinding</code>	CXF bean binding specified by the # notation. The referenced object must be an instance of <code>org.apache.camel.component.cxf.cxfbean.CxfBeanBinding</code> .	<code>cxfBinding=#bindingName</code>	No	DefaultC
<code>headerFilterStrategy</code>	Header filter strategy specified by the # notation. The referenced object must be an instance of <code>org.apache.camel.spi.HeaderFilterStrategy</code> .	<code>headerFilterStrategy=#strategyName</code>	No	CxfHeader
<code>populateFromClass</code>	Since 2.3, the <code>wsdlLocation</code> annotated in the POJO is ignored (by default) unless this option is set to <code>false</code> . Prior to 2.3, the <code>wsdlLocation</code> annotated in the POJO is always honored and it is not possible to ignore.	<code>true, false</code>	No	true
<code>providers</code>	Since 2.5, setting the providers for the CXFRS endpoint.	<code>providers=#providerRef1,#providerRef2</code>	No	null
<code>setDefaultBus</code>	Will set the default bus when CXF endpoint create a bus by itself.	<code>true, false</code>	No	false

Headers

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



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

A Working Sample

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

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

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.

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

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

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

URI format

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

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:



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.

Options

Name	Description	Example	Required?	default value
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>	No	None
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>	No	None
httpClientAPI	new to Camel 2.1 If it is true, the <code>CxfRsProducer</code> will use the <code>HttpClientAPI</code> to invoke the service If it is false, the <code>CxfRsProducer</code> will use the <code>ProxyClientAPI</code> to invoke the service	<code>httpClientAPI=true</code>	No	true
synchronous	New in 2.5, this option will let <code>CxfRsConsumer</code> 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.	<code>synchronous=true</code>	No	false
throwExceptionOnFailure	New in 2.6, this option tells the <code>CxfRsProducer</code> to inspect return codes and will generate an Exception if the return code is larger than 207.	<code>throwExceptionOnFailure=true</code>	No	true
maxClientCacheSize	New in 2.6, you can set a <code>LN</code> message header <code>CamelDestinationOverrideUrl</code> to dynamically override the target destination Web Service or REST Service defined in your routes. The implementation caches CXF clients or <code>ClientFactoryBean</code> in <code>CxfProvider</code> and <code>CxfRsProvider</code> . This option allows you to configure the maximum size of the cache.	<code>maxClientCacheSize=5</code>	No	10
setDefaultBus	New in 2.9.0. Will set the default bus when CXF endpoint create a bus by itself	<code>setDefaultBus=true</code>	No	false
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 <code>org.apache.cxf.Bus</code> .	<code>bus=#busName</code>	No	None
bindingStyle	As of 2.11 Sets how requests and responses will be mapped to/from Camel. Two values are possible: <ul style="list-style-type: none"> <code>SimpleConsumer</code> => see the Consuming a REST Request with the Simple Binding Style below. <code>Default</code> => the default style. For consumers this passes on a <code>MessageContentsList</code> to the route, requiring low-level processing in the route. 	<code>bindingStyle=SimpleConsumer</code>	No	Default

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 REST user guide 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.

cxfrsServer for REST consumer, **cxfrsClient** for REST producer.

You can find a Camel REST service route configuration example here.

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`:

```
-----
```

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:

```
-----
```

Serviced by the following route:

```
-----
```

The following HTTP request with XML payload (given that the Customer DTO is JAXB-annotated):

```
-----
```

Will print the message:

For more examples on how to process requests and write responses can be found [here](#).

Consuming a REST Request - Default Binding Style

CXF JAXRS front end implements the JAXRS(JSR311) 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`, you don't need to specify the URI template within your restlet endpoint, CXF take care of the REST request URI to resource class method mapping according to the JSR311 specification. All you need to do in Camel is delegate this method request to a right processor or endpoint.

Here is an example of a CXFRS route...

And the corresponding resource class used to configure the endpoint...

How to invoke the REST service through camel-cxfrs producer

CXF JAXRS front end implements a proxy based client API, with this API you can invoke the remote REST service through a proxy.

`camel-cxfrs` producer is based on this proxy API.

So, you just need to specify the operation name in the message header and prepare the parameter in the message body, `camel-cxfrs` producer will generate right REST request for you.

Here is an example

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 Http method and let the the producer know to use the http centric client by using the URI option **httpClientAPI** or set the message header with `CxfConstants.CAMEL_CXF_RS_USING_HTTP_API`. You can turn the response object to the type class that you specify with `CxfConstants.CAMEL_CXF_RS_RESPONSE_CLASS`.

From Camel 2.1, we also support to specify the query parameters from cxfrs URI for the CXFRS http centric client.

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.



note about the resource class

This class is used to configure the JAXRS properties ONLY. The methods will NOT be executed during the routing of messages to the endpoint, the route itself is responsible for ALL processing instead.

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

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 a default implementation, the `org.apache.camel.component.dataset.SimpleDataSet` that can be used for testing.

Options

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

You can append query options to the URI in the following format,

?option=value&option=value&...

Configuring DataSet

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

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:

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

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

Properties on SimpleDataSet

Property	Type	Default	Description
defaultBody	Object	<hello>world!</hello>	Specifies the default message body. For SimpleDataSet it is a constant payload; though if you want to create custom payloads per message, create your own derivation of DataSetSupport.
reportCount	long	-1	Specifies the number of messages to be received before reporting progress. Useful for showing progress of a large load test. If < 0, then size / 5, if is 0 then size, else set to reportCount value.
size	long	10	Specifies how many messages to send/consume.

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

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

Options

Name	Default Value	Description
<code>consumeDelete</code>	<code>true</code>	Option for <code>Db4oConsumer</code> only. Specifies whether or not the entity is deleted after it is consumed.
<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 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.

URI format



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**.

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

Options

Name	Default Value	Description
<code>allowMultipleConsumers</code>	<code>true</code>	@deprecated If set to <code>false</code> , 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.
<code>block</code>	<code>false</code>	Camel 2.11.1: If sending a message to a direct endpoint which has no active consumer, then we can tell the producer to block and wait for the consumer to become active.
<code>timeout</code>	<code>30000</code>	Camel 2.11.1: The timeout value to use if block is enabled.

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:

```
[...]
```

And the sample using spring DSL:

```
[...]
```

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

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

URI format

The URI scheme for a DNS component is as follows

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	*	lookup, dig	The type of the lookup. Should match the values of <code>org.xbill.dns.Type</code> . Optional.
dns.class	*	lookup, dig	he DNS class of the lookup. Should match the values of <code>org.xbill.dns.DClass</code> . 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

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"`.



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.

DNS lookup

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.

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:

URI format

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.
multiParameterArray	boolean	false	How to treat the parameters which are passed from the message body; if it is <code>true</code> , 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
```

And the implementation

```
Listing 1. GreaterImpl.java
```

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.

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

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:

Before we are ready to use EJB in the Camel routes:



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.

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

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

For example

Options

Name	Default Value	Description
<code>pattern</code>	<code>£</code>	The Esper Pattern expression as a String to filter events
<code>eql</code>	<code>£</code>	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:

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} Couldn't find a page to include called: Event

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

or

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 you own investigation what suites 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

Name	Default Value	Description
<code>autoCreate</code>	<code>true</code>	Automatically create missing directories in the file's pathname. For the file consumer, that means creating the starting directory. For the file producer, it means the directory the files should be written to.
<code>bufferSize</code>	<code>128kb</code>	Write buffer sized in bytes.
<code>fileName</code>	<code>null</code>	Use Expression such as File Language to dynamically set the filename. For consumers, it's used as a filename filter. For producers, it's used to evaluate the filename to write. If an expression is set, it take precedence over the <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. 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 onwards the producers support the <code>CamelOverrideFileName</code> header which takes precedence over any existing <code>CamelFileName</code> header; the <code>CamelOverrideFileName</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.
<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 <code>true</code> on the producer enforces that any file name recived 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, and camel will set the Exchange property with <code>Exchange.CHARSET_NAME</code> with the value of this option. 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 a examples and more important details.

copyAndDeleteOnRenameFail true

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.

Consumer

Name	Default Value	Description
initialDelay	1000	Milliseconds before polling the file/directory starts.
delay	500	Milliseconds before the next poll of the file/directory.
useFixedDelay	£	Controls if fixed delay or fixed rate is used. See <code>ScheduledExecutorService</code> in JDK for details. In Camel 2.7.x or older the default value is <code>false</code> . From Camel 2.8 onwards the default value is <code>true</code> .
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.
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
noop	false	If true, the file is not moved or deleted in any way. This option is good for readonly data, or for ETL type requirements. If <code>noop=true</code> , Camel will set <code>idempotent=true</code> 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 when moving it before processing. For example to move in-progress files into the <code>order</code> directory set this value to <code>order</code> .
move	.camel	Expression (such as File Language) used to dynamically set the filename when moving it after processing. To move files into a <code>.done</code> subdirectory just enter <code>.done</code> .
moveFailed	null	Expression (such as File Language) used to dynamically set a different target directory when moving files after processing (configured via <code>move</code> defined above) failed. For example, to move files into a <code>.error</code> subdirectory use <code>.error</code> . Note: When moving the files to the <code>failO</code> location Camel will handle the error and will not pick up the file again.
include	null	Is used to include files, if filename matches the regex pattern.
exclude	null	Is used to exclude files, if filename matches the regex pattern.
antInclude	null	Camel 2.10: Ant style filter inclusion, for example <code>antInclude=*.txt</code> . Multiple 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 <code>antInclude</code> and <code>antExclude</code> are used, <code>antExclude</code> takes precedence over <code>antInclude</code> . Multiple exclusions may 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 already processed files. Will by default use a memory based <code>LRUCache</code> that holds 1000 entries. If <code>noop=true</code> then idempotent will be enabled as well to avoid consuming the same files over and over again.
idempotentKey	Expression	Camel 2.11: To use a custom idempotent key. By default the absolute path of the file is used. You can use the File Language, for example to use the file name and file size, you can do: <div><pre>file:name size</pre></div>
idempotentRepository	null	A pluggable repository <code>org.apache.camel.spi.IdempotentRepository</code> which by default use <code>MemoryMessageIdRepository</code> if none is specified and <code>idempotent</code> is <code>true</code> .
inProgressRepository	memory	A pluggable in-progress repository <code>org.apache.camel.spi.IdempotentRepository</code> . 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 <code>org.apache.camel.component.file.GenericFileFilter</code> class. Will skip files if filter returns <code>false</code> in its <code>accept()</code> method. More details in section below.
sorter	null	Pluggable sorter as a <code>java.util.Comparator<org.apache.camel.component.file.GenericFile></code> class.
sortBy	null	Built-in sort using the File Language. Supports nested sorts, so you can have a sort by file name and as a 2nd group sort by modified date. See sorting section below for details.

		<p>Used by consumer, to only poll the files if it has exclusive read-lock on the file (i.e. the file is not in-progress or being written). Camel will wait until the file lock is granted.</p> <p>This option provides the build in strategies:</p> <p><code>markerFile</code> Camel creates a marker file and then holds a lock on it. This option is not available for the FTP component.</p> <p><code>changed</code> is using file length/modification timestamp to detect whether the file is currently being copied or not. Will at least use 1 sec. to determine this, so this option cannot consume files as fast as the others, but can be more reliable as the JDK IO API cannot always determine whether a file is currently being used by another process. The option <code>readLockCheckInterval</code> can be used to set the check frequency. This option is only avail for the FTP component from Camel 2.8 onwards. Notice that from Camel 2.10.1 onwards the FTP option <code>fastExistsCheck</code> can be enabled to speedup this readLock strategy, if the FTP server support the LIST operation with a full file name (some servers may not).</p> <p><code>fileLock</code> is for using <code>java.nio.channels.FileLock</code>. This option is not avail for the FTP component. This approach should be avoided when accessing a remote file system via a mount/share unless that file system supports distributed file locks.</p> <p><code>rename</code> is for using a try to rename the file as a test if we can get exclusive read-lock.</p> <p><code>none</code> is for no read locks at all.</p> <p>Notice from Camel 2.10 onwards the read locks <code>changed</code>, <code>fileLock</code> and <code>rename</code> will also use a <code>markerFile</code> as well, to ensure not picking up files that may be in process by another Camel consumer running on another node (eg cluster). This is only supported by the file component (not the ftp component).</p>
<code>readLock</code>	<code>markerFile</code>	
<code>readLockTimeout</code>	10000	<p>Optional timeout in millis for the read-lock, if supported by the read-lock. If the read-lock could not be granted and the timeout triggered, then Camel will skip the file. At next 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 <code>fileLock</code>, <code>changed</code> and <code>rename</code> support the timeout. Notice: For FTP the default <code>readLockTimeout</code> value is 20000 instead of 10000.</p>
<code>readLockCheckInterval</code>	1000	<p>Camel 2.6: Interval in millis for the read-lock, if supported by the read lock. This interval is used for sleeping between attempts to acquire the read lock. For example when using the <code>changed</code> read lock, you can set a higher interval period to cater for <i>slow writes</i>. The default of 1 sec. may be <i>too fast</i> if the producer is very slow writing the file. For FTP the default <code>readLockCheckInterval</code> is 5000.</p>
<code>readLockMinLength</code>	1	<p>Camel 2.10.1: This option applied only for <code>readLock=changed</code>. This option allows you to configure a minimum file length. By default Camel expects the file to contain data, and thus the default value is 1. You can set this option to zero, to allow consuming zero-length files.</p>
<code>readLockLoggingLevel</code>	WARN	<p>Camel 2.12: Logging level used when a read lock could not be acquired. By default a WARN is logged. You can change this level, for example to OFF to not have any logging. This option is only applicable for readLock of types: <code>changed</code>, <code>fileLock</code>, <code>rename</code>.</p>
<code>directoryMustExist</code>	false	<p>Camel 2.5: Similar to <code>startingDirectoryMustExist</code> but this applies during polling recursive sub directories.</p>
<code>doneFileName</code>	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 name. Or you can use dynamic placeholders. The <i>done</i> file is always expected in the same folder as the original file. See <i>using done file</i> and <i>writing done file</i> sections for examples.</p>
<code>exclusiveReadLockStrategy</code>	null	<p>Pluggable read-lock as a <code>org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy</code> implementation.</p>
<code>maxMessagesPerPoll</code>	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 starting up the server that there are thousands of files. Set a value of 0 or negative to disabled it. See more details at Batch Consumer. Notice: If this option is in use then the File and FTP components will limit before any sorting. For example if you have 100000 files and use <code>maxMessagesPerPoll=500</code>, then only the first 500 files will be picked up, and then sorted. You can use the <code>eagerMaxMessagesPerPoll</code> option and set this to <code>false</code> to allow to scan all files first and then sort afterwards.</p>
<code>eagerMaxMessagesPerPoll</code>	true	<p>Camel 2.9.3: Allows for controlling whether the limit from <code>maxMessagesPerPoll</code> is eager or not. If eager then the limit is during the scanning of files. Where as <code>false</code> would scan all files, and then perform sorting. Setting this option to <code>false</code> allows for sorting all files first, and then limit the poll. Mind that this requires a higher memory usage as all file details are in memory to perform the sorting.</p>
<code>minDepth</code>	0	<p>Camel 2.8: The minimum depth to start processing when recursively processing a directory. Using <code>minDepth=1</code> means the base directory. Using <code>minDepth=2</code> means the first sub directory. This option is supported by FTP consumer from Camel 2.8.2, 2.9 onwards.</p>
<code>maxDepth</code>	<code>Integer.MAX_VALUE</code>	<p>Camel 2.8: The maximum depth to traverse when recursively processing a directory. This option is supported by FTP consumer from Camel 2.8.2, 2.9 onwards.</p>
<code>processStrategy</code>	null	<p>A pluggable <code>org.apache.camel.component.file.GenericFileProcessStrategy</code> allowing you to implement your own <code>readLock</code> option or similar. Can also be used when special conditions must be met before a file can be consumed, such as a special <i>ready</i> file exists. If this option is set then the <code>readLock</code> option does not apply.</p>
<code>startingDirectoryMustExist</code>	false	<p>Camel 2.5: Whether the starting directory must exist. Mind that the <code>autoCreate</code> option is default enabled, which means the starting directory is normally auto created if it doesn't exist. You can disable <code>autoCreate</code> and enable this to ensure the starting directory must exist. Will throw an exception if the directory doesn't exist.</p>

<code>pollStrategy</code>	<code>null</code>	A pluggable <code>org.apache.camel.PollingConsumerPollStrategy</code> allowing you to provide your custom implementation to control error handling usually occurred during the <code>poll</code> operation before an Exchange have been created and being 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 <code>WARN</code> level and ignore it.
<code>sendEmptyMessageWhenIdle</code>	<code>false</code>	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.
<code>consumer.bridgeErrorHandler</code>	<code>false</code>	Camel 2.10: Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while trying to pickup files, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the <code>org.apache.camel.spi.ExceptionHandler</code> to deal with exceptions, that by default will be logged at <code>WARN/ERROR</code> level and ignored. See further below on this page for more details, at section <i>How to use the Camel error handler to deal with exceptions triggered outside the routing engine</i> .
<code>scheduledExecutorService</code>	<code>null</code>	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.
<code>scheduler</code>	<code>null</code>	Camel 2.12: To use a custom scheduler to trigger the consumer to run. See more details at <i>Polling Consumer</i> , for example there is a <code>Quartz2</code> , and <code>Spring</code> based scheduler that supports <code>CRON</code> expressions.
<code>backoffMultiplier</code>	<code>0</code>	Camel 2.12: To let the scheduled polling consumer backoff 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. See more details at <i>Polling Consumer</i> .
<code>backoffIdleThreshold</code>	<code>0</code>	Camel 2.12: The number of subsequent idle polls that should happen before the <code>backoffMultiplier</code> should kick-in.
<code>backoffErrorThreshold</code>	<code>0</code>	Camel 2.12: The number of subsequent error polls (failed due some error) that should happen before the <code>backoffMultiplier</code> should kick-in.

Default behavior for file consumer

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

Producer

Name	Default Value	Description
<code>fileExist</code>	<code>Override</code>	What to do if a file already exists with the same name. The following values can be specified: Override , Append , Fail , Ignore , Move , and TryRename (Camel 2.11.1). Override , which is the default, replaces the existing file. Append adds content to the existing file. Fail throws a <code>GenericFileOperationException</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. The Move option requires Camel 2.10.1 onwards, and the corresponding <code>moveExisting</code> option to be configured as well. The option <code>eagerDeleteTargetFile</code> can be used to control what to do if an moving the file, and there exists already an existing 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 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.
<code>tempPrefix</code>	<code>null</code>	This option is used to write the file using a temporary name and then, after the write is complete, rename it to the real name. Can be used to identify files being written and also avoid consumers (not using exclusive read locks) reading in progress files. Is often used by FTP when uploading big files.
<code>tempFileName</code>	<code>null</code>	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.

moveExisting	null	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 <code>backup</code> subdirectory just enter <code>backup</code> . This option only supports the following File Language tokens: "file:name", "file:name.ext", "file:name.noext", "file:onlyname", "file:onlyname.noext", "file:ext", and "file:parent". Notice the "file:parent" is not supported by the FTP component, as the FTP component can only move any existing files to a relative directory based on current dir as base.
keepLastModified	false	Camel 2.2: Will keep the last modified timestamp from the source file (if any). Will use the <code>Exchange.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. Note: This option only applies to the file producer. You <i>cannot</i> use this option with any of the ftp producers.
eagerDeleteTargetFile	true	Camel 2.3: Whether or not to eagerly delete any existing target file. This option only applies when you use <code>fileExists=Override</code> and the <code>tempFileName</code> option as well. You can use this to disable (set it to false) deleting the target file before the temp file is written. For example you may write big files and want the target file to exists during the temp file is being written. This ensure the target file is only deleted until the very last moment, just before the temp file is being renamed to the target filename. From Camel 2.10.1 onwards 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 is false, then an exception will be thrown if an existing file existed, if its true, then the existing file is deleted before the move operation.
doneFileName	null	Camel 2.6: If provided, then Camel will write a 2nd <i>done</i> file when the original file has been written. The <i>done</i> file will be empty. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders. The <i>done</i> file will always be written in the same folder as the original file. See <i>writing 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 <code>GenericFileWriteException</code> of 'Cannot write null body to file.' will be thrown. If the 'fileExist' option is set to 'Override', then the file will be truncated, and if set to '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.

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:

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 is 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 - eg if you set

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:

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.

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

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

Fine grained control over Move and PreMove option

The **move** and **preMove** option is Expression-based, so we have the full power of the File Language 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:

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

Header	Description
--------	-------------

CamelFileName	Specifies the name of the file to write (relative to the endpoint directory). The name can be a <code>String</code> ; a <code>String</code> with a File Language or Simple expression; or an Expression object. If it's <code>null</code> then Camel will auto-generate a filename based on the message unique ID.
CamelFileNameProduced	The actual absolute filepath (path + name) for the output file that was written. This header is set by Camel and its purpose is providing end-users with the name of the file that was written.
CamelOverrideFileName	Camel 2.11: Is used for overruling <code>CamelFileName</code> 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 <code>String</code> . Notice that if the option <code>fileName</code> has been configured, then this is still being evaluated.

File consumer only

Header	Description
CamelFileName	Name of the consumed file as a relative file path with offset from the starting directory configured on the endpoint.
CamelFileNameOnly	Only the file name (the name with no leading paths).
CamelFileAbsolute	A <code>boolean</code> option specifying whether the consumed file denotes an absolute path or not. Should normally be <code>false</code> 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 <code>long</code> value containing the file size.
CamelFileLastModified	A <code>Date</code> value containing the last modified timestamp of the file.

Batch Consumer

This component implements the Batch Consumer.

Exchange Properties, file consumer only

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

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

This allows you for instance to know how many files exists 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:

```

// ...

```

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.

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).

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.

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:

And the logs:

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:

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

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

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

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

Using done files

Available as of Camel 2.6

See also section *writing done files* below.

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

Will only consume files from the bar folder, if a file name `done` 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** onwards Camel will not automatic delete the done file if `noop=true` is configured.

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

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:

- `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 kinda of marker, to indicate to others that the file is finished and has been written. To do that you can use the `doneFileName` option on the file producer endpoint.

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

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

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.

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.

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

Samples

Read from a directory and write to another directory

Read from a directory and write to another directory using a overrule dynamic 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

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.

Will result in the following output layout:

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:

Will result in the following output layout:

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.

Affects the layout as follows:

before

after

Read from a directory and process the message in java

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 written to a directory.

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:

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

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

Using expression for filenames

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

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.

Camel uses the absolute file name as the idempotent key, to detect duplicate files. From **Camel 2.11** onwards 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

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 plugin your own implementation of this store by using the `idempotentRepository` option using the `#` sign in the value to indicate it's a referring to a bean in the Registry with the specified `id`.

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

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:

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.
```

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

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

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

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:

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:

Filtering using ANT path matcher

The ANT path matcher is shipped out-of-the-box in the **camel-spring** jar. So you need to depend on **camel-spring** if you are using Maven.

The 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



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.

- `**` matches zero or more directories in a path

The sample below demonstrates how to use it:

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:

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:

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:

Where each group is separated with semi colon. In the simple situations you just use one group, so a simple example could be:

This will sort by file name, you can reverse the order by prefixing `reverse:` to the group, so the sorting is now Z..A:

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:

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:

You can combine ignore case and reverse, however reverse must be specified first:

In the sample below we want to sort by last modified file, so we do:



URI options can reference beans using the # syntax

In the Spring DSL route about 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`.

And then we want to group by name as a 2nd option so files with same modification is sorted by 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 the its date command that supports patterns. So this can be solved as:

Yeah, that is pretty powerful, oh by the way you can also use reverse per group, so we could reverse the file names:

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 onwards, 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:

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 (eg 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.

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":

Listing 1. MyExceptionHandler

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"/>`:

Listing 1. Camel route with routing engine error handling

The source code for this example can be seen here



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.

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:

Listing 1. Using `consumer.bridgeErrorHandler`

So all you have to do is to enable this option, and the error handler in the route will take it from there.

Debug logging

This component has log level **TRACE** that can be helpful if you have problems.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- 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:

URI format



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.

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

```
-----
```

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>splitRows</code>	<code>true</code>	The component can either process each row one by one or the entire content at once.
<code>allowShortLines</code>	<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.
<code>ignoreExtraColumns</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>splitRows=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:

However, you can also always get it as a `List` (even for `splitRows=true`). The same example:

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.

Using the endpoint

A common use case is sending a file to this endpoint for further processing in a separate route. For example:

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	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:

```
routeBuilder().unmarshal().flatpack().marshal().flatpack();
```

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.

```
routeBuilder().unmarshal().flatpack().marshal().flatpack().to("seda:queue1");
```

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).

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:

URI format

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

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:

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:

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 for InOut message exchange pattern).

Hot reloading

The FreeMarker template resource is by default **not** hot reloadable for both file and classpath resources (expanded jar). If you set `contentCache=false`, then 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.

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 to use instead of the endpoint configured.	>= 2.1

Samples

For example you could use something like:

```
.....
```

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:

```
.....
```

And to disable the content cache, e.g. for development usage where the `.ftl` template should be hot reloaded:

```
.....
```

And a file-based resource:

In **Camel 2.1** it's possible to specify what template the component should use dynamically via a header, so for example:

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:

And the java code:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

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:

URI format

Where **directoryname** represents the underlying directory. Can contain nested folders.

If no **username** is provided, then `anonymous` login is attempted using no password.

If no **port** number is provided, 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.



More options

See File for more options as all the options from File is inherited.



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 Options

The options below are exclusive for the FTP component.

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.
binary	false	Specifies the file transfer mode, BINARY or ASCII. Default is ASCII (<code>false</code>).
disconnect	false	Camel 2.2: Whether or not to disconnect from remote FTP server right after use. Can be used for both consumer and producer. Disconnect will only disconnect the current connection to the FTP server. If you have a consumer which you want to stop, then you need to stop the consumer/route instead.
localWorkDirectory	null	When consuming, a local work directory can be used to store the remote file content directly in local files, to avoid loading the content into memory. This is beneficial, if you consume a very big remote file and thus can conserve memory. See below for more details.
passiveMode	false	FTP and FTPS only: Specifies whether to use passive mode connections. Default is active mode (<code>false</code>).
securityProtocol	TLS	FTPS only: Sets the underlying security protocol. The following values are defined: TLS: Transport Layer Security SSL: Secure Sockets Layer
disableSecureDataChannelDefaults	false	Camel 2.4: FTPS only: Whether or not to disable using default values for <code>execPbsz</code> and <code>execProt</code> when using secure data transfer. You can set this option to <code>true</code> if you want to be in absolute full control what the options <code>execPbsz</code> and <code>execProt</code> should be used.
download	true	Camel 2.1.1: Whether the FTP consumer should download the file. If this option is set to <code>false</code> , then the message body will be <code>null</code> , but the consumer will still trigger a Camel Exchange that has details about the file such as file name, file size, etc. It's just that the file will not be downloaded.
streamDownload	false	Camel 2.1.1: Whether the consumer should download the entire file up front, the default behavior, or if it should pass an <code>InputStream</code> read from the remote resource rather than an in-memory array as the in body of the <code>CamelExchange</code> . This option is ignored if <code>download</code> is <code>false</code> or is <code>localWorkDirectory</code> is provided. This option is useful for working with large remote files.
execProt	null	Camel 2.4: FTPS only: Will by default use option <code>P</code> if secure data channel defaults hasn't been disabled. Possible values are: C: Clear S: Safe (SSL protocol only) E: Confidential (SSL protocol only) P: Private
execPbsz	null	Camel 2.4: FTPS only: This option specifies the buffer size of the secure data channel. If option <code>useSecureDataChannel</code> has been enabled and this option has not been explicit set, then value <code>0</code> is used.
isImplicit	false	FTPS only: Sets the security mode(implicit/explicit). Default is explicit (<code>false</code>).
knownHostsFile	null	SFTP only: Sets the <code>known_hosts</code> file, so that the SFTP endpoint can do host key verification.
knownHostsUri	null	SFTP only: Camel 2.11.1: Sets the <code>known_hosts</code> file (loaded from classpath by default), so that the SFTP endpoint can do host key verification.
keyPair	null	SFTP only: Camel 2.12.0: Sets the Java <code>KeyPair</code> for SSH public key authentication, it supports DSA or RSA keys.
privateKeyFile	null	SFTP only: Set the private key file to that the SFTP endpoint can do private key verification.



More options

See File for more options as all the options from File is inherited.

privateKeyUri	null	SFTP only; Camel 2.11.1: Set the private key file (loaded from classpath by default) to 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 authentications which SFTP endpoint will used. Some example include: password, publickey. If not specified the default list from JSCH will be used.
ciphers	null	Camel 2.8.2, 2.9; SFTP only Set a comma separated list of ciphers that will be used in order of preference. Possible cipher names are defined by JCraft JSCH. Some examples include: aes128-ctr,aes128-cbc,3des-ctr,3des-cbc,blowfish-cbc,aes192-cbc,aes256-cbc. If not specified the default list from JSCH will be used.
fastExistsCheck	false	Camel 2.8.2, 2.9; If set this option to be true, camel-ftp will use the list file directly to check if the file exists. Since some FTP server may not support to list the file directly, if the option is false, camel-ftp will use the old way to list the directory and check if the file exists. Note from Camel 2.10.1 onwards this option also influences readLock=changed to control whether it performs a fast check to update file information or not. This can be used to 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. Possible values are: no, yes and ask. ask does not make sense to use as Camel cannot answer the question for you as its meant for human intervention. Note: The default in Camel 2.1 and below was ask.
maximumReconnectAttempts	3	Specifies the maximum reconnect attempts Camel performs when it tries to connect to the remote FTP server. Use 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 ftpClient.connectTimeout for the FTP/FTPS. For SFTP this option is also used when attempting to connect.
soTimeout	null	FTP and FTPS Only; Camel 2.4: Is the SocketOptions.SO_TIMEOUT value in millis. Note SFTP will automatic use the connectTimeout as the soTimeout.
timeout	30000	FTP and FTPS Only; Camel 2.4: Is the data timeout in millis. This corresponds to using ftpClient.dataTimeout for the FTP/FTPS. For SFTP there is no data timeout.
throwExceptionOnConnectFailed	false	Camel 2.5: Whether or not to thrown an exception if a successful connection and login could not be establish. This allows a custom pollStrategy to deal with the exception, for example to stop the consumer or the likes.
siteCommand	null	FTP and FTPS Only; Camel 2.5: To execute site commands after successful login. Multiple site commands can be separated using a new line character (\n). Use help site to see which site commands your FTP server supports.
stepwise	true	Camel 2.6: Whether or not stepwise traversing directories should be used or not. Stepwise means that it will CD one directory at a time. See more details below. You can disable this in case you can't use this approach.
separator	Auto	Camel 2.6: Dictates what path separator char to use when uploading files. Auto = Use the path provided without altering it. UNIX = Use unix style path separators. Windows = Use Windows style path separators.
chmod	null	SFTP Producer Only; Camel 2.9: Allows you to set chmod on the stored file. For example chmod=640.
compression	0	SFTP Only; Camel 2.8.3/2.9: To use compression. Specify a level from 1 to 10. Important: You must manually add the needed JSCH zlib JAR to the classpath for compression support.
ftpClient	null	FTP and FTPS Only; Camel 2.1: Allows you to use a custom org.apache.commons.net.ftp.FTPClient instance.
ftpClientConfig	null	FTP and FTPS Only; Camel 2.1: Allows you to use a custom org.apache.commons.net.ftp.FTPClientConfig instance.
serverAliveInterval	0	SFTP Only; Camel 2.8 Allows you to set the serverAliveInterval of the sftp session
serverAliveCountMax	1	SFTP Only; Camel 2.8 Allows you to set the serverAliveCountMax of the sftp session
ftpClient.trustStore.file	null	FTPS Only: Sets the trust store file, so that the FTPS client can look up for trusted certificates.

<code>ftpClient.trustStore.type</code>	JKS	FTPS Only: Sets the trust store type.
<code>ftpClient.trustStore.algorithm</code>	SunX509	FTPS Only: Sets the trust store algorithm.
<code>ftpClient.trustStore.password</code>	null	FTPS Only: Sets the trust store password.
<code>ftpClient.keyStore.file</code>	null	FTPS Only: Sets the key store file, so that the FTPS client can look up for the private certificate.
<code>ftpClient.keyStore.type</code>	JKS	FTPS Only: Sets the key store type.
<code>ftpClient.keyStore.algorithm</code>	SunX509	FTPS Only: Sets the key store algorithm.
<code>ftpClient.keyStore.password</code>	null	FTPS Only: Sets the key store password.
<code>ftpClient.keyStore.keyPassword</code>	null	FTPS Only: Sets the private key password.
<code>sslContextParameters</code>	null	FTPS Only: Camel 2.9: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry. This reference overrides any configured SSL related options on <code>ftpClient</code> as well as the securityProtocol (SSL, TLS, etc.) set on <code>FtpsConfiguration</code> . See Using the JSSE Configuration Utility.
<code>proxy</code>	null	SFTP Only: Camel 2.10.7, 2.11.1: Reference to a <code>com.jcraft.jsch.Proxy</code> in the Registry. This proxy is used to consume/send messages from the target SFTP host.
<code>useList</code>	true	FTP/FTPS Only: Camel 2.12.1: Whether the consumer should use FTP LIST command to retrieve directory listing to see which files exists. If this option is set to <code>false</code> , then <code>stepwise=false</code> must be configured, and also <code>fileName</code> must be configured to a fixed name, so the consumer knows the name of the file to retrieve. When doing this only that single file can be retrieved. See further below for more details.
<code>ignoreFileNotFoundOrPermissionError</code>	false	Camel 2.12.1: Whether the consumer should ignore when a file was attempted to be retrieved but did not exist (for some reason), or failure due insufficient file permission error.

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:

```
ftpClient.setDataTimeout=30
```

You can mix and match and have use both prefixes, for example to configure date format or timezones.

```
ftpClientConfig.dateFormat=dd/MM/yyyy&ftpClientConfig.timezone=GMT+01:00
```

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:

```
ftpClient=#lookup&ftpClientConfig=#lookup
```

And then let Camel lookup this bean when you use the `#` notation in the url.

```
ftp://someone@someftpserver.com/public/upload/images/holiday2008?password=secret&binary=true&ftpClient=#lookup&ftpClientConfig=#lookup
```

More URI options

Examples

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



FTPS component default trust store

When using the `ftpClient.` properties related to SSL with the FTPS component, the trust store accept all certificates. If you only want trust selective certificates, you have to configure the trust store with the `ftpClient.trustStore.xxx` options or by configuring a custom `ftpClient`.

When using `sslContextParameters`, the trust store is managed by the configuration of the provided `SSLContextParameters` instance.



See File2 as all the options there also applies for this component.

```
password=secret&binary=false  
ftp://publicftpserver.com/download
```

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 is currently in the progress of 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.



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.

Message Headers

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

Header	Description
CamelFileName	Specifies the output file name (relative to the endpoint directory) to be used for the output message when sending to the endpoint. If this is not present and no expression either, then a generated message ID is used as the filename instead.
CamelFileNameProduced	The actual absolute filepath (path + name) for the output file that was written. This header is set by Camel and its purpose is providing end-users the name of the file that was written.
CamelFileBatchIndex	Current index out of total number of files being consumed in this batch.
CamelFileBatchSize	Total number of files being consumed in this batch.
CamelFileHost	The remote hostname.
CamelFileLocalWorkPath	Path to the local work file, if local work directory is used.

In addition the FTP/FTPS consumer and producer will enrich the Camel Message with the following headers

Header	Description
CamelFtpReplyCode	Camel 2.11.1: The FTP client reply code (the type is a integer)
CamelFtpReplyString	Camel 2.11.1: The FTP client reply string

About timeouts

The two set of libraries (see top) has different API 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.setSoTimeout`. 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.setDataTimeout` 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:

```
file:local-work-dir?noop=true
```

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 it's home directory and when the home directory is reported as `" / "`.

The difference between the two of them is best illustrated with an example. Suppose we have the following directory structure on the remote FTP server we need to traverse and download files:

```
ftp://remote-server.com/remote-dir/
```

And that we have a file in each of sub-a (a.txt) and sub-b (b.txt) folder.

Using `stepwise=true` (default mode)

```
ftp:ftp://remote-server.com/remote-dir?stepwise=true
```

As you can see when stepwise is enabled, it will traverse the directory structure using `CD xxx`.

Using `stepwise=false`

```
ftp:ftp://remote-server.com/remote-dir?stepwise=false
```



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.

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.

And the route using Spring DSL:

Consuming a remote FTPS server (implicit SSL) and client authentication

Consuming a remote FTPS server (explicit TLS) and a custom trust store configuration

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

Camel supports pluggable filtering strategies. This strategy it to use the build in `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.

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:

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:

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:

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.

Consuming a single file using a fixed name

When you want to download a single file and knows 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 `ignoreFileNotFoundOrPermissionError=true`.

For example to have a Camel route that pickup a single file, and delete it after use you can do

Notice that we have use all the options we talked above 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:

Debug logging

This component has log level **TRACE** that can be helpful if you have problems.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [File2](#)

CAMEL COMPONENTS FOR GOOGLE APP ENGINE

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	ghhttp	Provides connectivity to the GAE URL fetch service but can also be used to receive messages from servlets.
Task queueing service	gtask	Supports asynchronous message processing on GAE by using the task queueing 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	Ê	Not supported yet.
XMPP service	Ê	Not supported yet.
Images service	Ê	Not supported yet.
Datastore service	Ê	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 GAE applications 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.



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.

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

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

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

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:

```
<code></code>
```

URI format

```
<code></code>
```

Sections

1. Usage of map
2. Usage of multimap
3. Usage of queue
4. Usage of list
5. Usage of seda
6. Usage of atomic number
7. Usage of cluster support (instance)

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.
```



You have to use the second prefix to define which type of data store you want to use.

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)

Name	Type	Description
CamelHazelcastOperationType	String	valid values are: put, delete, get, update, query [Version 2.8]
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:

Sample for put:

Java DSL:

Spring DSL:

Sample for get:

Java DSL:

Spring DSL:

Sample for update:

Java DSL:



Header variables have changed in Camel 2.8

Spring DSL:

```
-----
```

Sample for delete:

Java DSL:

```
-----
```

Spring DSL:

```
-----
```

Sample for query

Java DSL:

```
-----
```

Spring DSL:

```
-----
```

For the query operation Hazelcast offers a SQL like syntax to query your distributed map.

```
-----
```

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 **evict**. 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 "cachelister"
hazelcast.listener.action	String	type of event - here added , updated , evicted 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 "cachelister" [Version 2.8]
CamelHazelcastListenerAction	String	type of event - here added, updated, evicted 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:

```

<?xml version="1.0" ?>
<body>
  <put>
    <value>
      <map>
        <entry>
          <key>key</key>
          <value>value</value>
        </entry>
      </map>
    </value>
  </put>
  <update>
    <value>
      <map>
        <entry>
          <key>key</key>
          <value>value</value>
        </entry>
      </map>
    </value>
  </update>
</body>

```

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
CamelHazelcastOperationType	String	valid values are: put, delete, get, update, query Available as of Camel 2.8
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:

[.....]

Sample for put:

Java DSL:

[.....]

Spring DSL:

[.....]

Sample for get:

Java DSL:

[.....]

Spring DSL:

[.....]

Sample for update:

Java DSL:

[.....]

Spring DSL:

[.....]

Sample for delete:

Java DSL:

[.....]

Spring DSL:

[.....]

Sample for query

Java DSL:

```
-----
```

Spring DSL:

```
-----
```

For the query operation Hazelcast offers a SQL like syntax to query your distributed map.

```
-----
```

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 **evict**. 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 "cachelister"
hazelcast.listener.action	String	type of event - here added , updated , evicted 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

Name	Type	Description
CamelHazelcastListenerTime	Long	time of the event in millis [Version 2.8]
CamelHazelcastListenerType	String	the map consumer sets here "cachelister" [Version 2.8]
CamelHazelcastListenerAction	String	type of event - here added , updated , evicted 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]



Header variables have changed in Camel 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:

```

-----

```

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

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:

```

-----

```

Spring DSL:

```

-----

```




Header variables have changed in Camel 2.8

Sample for removevalue:

Java DSL:

```
.....
```

Spring DSL:

```
.....
```

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:

```
.....
```

Spring DSL:

```
.....
```

Sample for delete:

Java DSL:

```
.....
```

Spring DSL:

```
.....
```

you can call them in your test class with:

```
.....
```

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:

```
.....
```

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 "cachelister"

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).

Name	Type	Description
CamelHazelcastListenerTime	Long	time of the event in millis [Version 2.8]
CamelHazelcastListenerType	String	the map consumer sets here "cachelister" [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:

```
queue.add("foo")
```

Sample for put:

```
queue.put("foo")
```

Sample for poll:

```
queue.poll()
```



Header variables have changed in Camel 2.8

Sample for peek:

Sample for offer:

Sample for removevalue:

Queue consumer **to** from(`Øhazelcast:queue:fooÓ`)

The queue consumer provides 2 operations (add, remove).

Usage of List

List producer **to** (`Øhazelcast:list:fooÓ`)

The list producer provides 4 operations (add, set, get, removevalue).

Sample for add:

Sample for get:

Sample for setvalue:

Sample for removevalue:



Please note that set,get and removevalue and not yet supported by hazelcast, will be added in the future..

List consumer Ð from(Òhazelcast:list:fooÓ)

The list consumer provides 2 operations (add, remove).

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 Ð to(Òhazelcast:seda:fooÓ)

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 transfered. If header or body contains not serializable objects, they will be skipped.

Java DSL :

Spring DSL :

SEDA consumer Ð from(Òhazelcast:seda:fooÓ)

The SEDA consumer provides no operations. You only retrieve data from the specified queue.

Name	default value	Description
pollInterval	1000	How frequent to poll from the SEDA queue
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 transfered. 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 :

```
.....
```

Spring DSL:

```
.....
```

Usage of Atomic Number

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

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:

```
.....
```

Spring DSL:

```
.....
```

Provide the value to set inside the message body (here the value is 10):

```
template.sendBody("direct:set", 10);
```

Sample for get:

Java DSL:

```
.....
```



There is no consumer for this endpoint!



Header variables have changed in Camel 2.8

Spring DSL:

```
template.requestBody("direct:get", null, Long.class);
```

You can get the number with `long body =`

```
template.requestBody("direct:get", null, Long.class);
```

Sample for increment:

Java DSL:

```
template.requestBody("direct:get", null, Long.class);
```

Spring DSL:

```
template.requestBody("direct:get", null, Long.class);
```

The actual value (after increment) will be provided inside the message body.

Sample for decrement:

Java DSL:

```
template.requestBody("direct:get", null, Long.class);
```

Spring DSL:

```
template.requestBody("direct:get", null, Long.class);
```

The actual value (after decrement) will be provided inside the message body.

Sample for destroy

Java DSL:

```
template.requestBody("direct:get", null, Long.class);
```

Spring DSL:

```
template.requestBody("direct:get", null, Long.class);
```

cluster support

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.



There's a bug inside Hazelcast. So this feature may not work properly. Will be fixed in 1.9.3.



This endpoint provides no producer!

Here's a sample:

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

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]

HDFS COMPONENT

Available as of Camel 2.8



Header variables have changed in 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:

```
<code></code>
```

URI format

```
<code></code>
```

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

The path is treated in the following way:

1. as a consumer, if it's a file, it just reads the file, otherwise if it represents a directory it scans all the file under the path satisfying the configured pattern. All the files under that directory must be of the same type.
2. as a producer, if at least one split strategy is defined, the path is considered a directory and under that directory the producer creates a different file per split named `seg0`, `seg1`, `seg2`, etc.

Options

Name	Default Value	Description
<code>overwrite</code>	<code>true</code>	The file can be overwritten
<code>append</code>	<code>false</code>	Append to existing file. Notice that not all HDFS file systems support the append option.
<code>bufferSize</code>	<code>4096</code>	The buffer size used by HDFS
<code>replication</code>	<code>3</code>	The HDFS replication factor
<code>blockSize</code>	<code>67108864</code>	The size of the HDFS blocks
<code>fileType</code>	<code>NORMAL_FILE</code>	It can be <code>SEQUENCE_FILE</code> , <code>MAP_FILE</code> , <code>ARRAY_FILE</code> , or <code>BLOOMMAP_FILE</code> , see Hadoop
<code>fileSystemType</code>	<code>HDFS</code>	It can be <code>LOCAL</code> for local filesystem
<code>keyType</code>	<code>NULL</code>	The type for the key in case of sequence or map files. See below.
<code>valueType</code>	<code>TEXT</code>	The type for the key in case of sequence or map files. See below.
<code>splitStrategy</code>	<code>É</code>	A string describing the strategy on how to split the file based on different criteria. See below.
<code>openedSuffix</code>	<code>opened</code>	When a file is opened for reading/writing the file is renamed with this suffix to avoid to read it during the writing phase.
<code>readSuffix</code>	<code>read</code>	Once the file has been read is renamed with this suffix to avoid to read it again.
<code>initialDelay</code>	<code>0</code>	For the consumer, how much to wait (milliseconds) before to start scanning the directory.
<code>delay</code>	<code>0</code>	The interval (milliseconds) between the directory scans.
<code>pattern</code>	<code>*</code>	The pattern used for scanning the directory
<code>chunkSize</code>	<code>4096</code>	When reading a normal file, this is split into chunks producing a message per chunk.
<code>connectOnStartup</code>	<code>true</code>	Camel 2.9.3/2.10.1: Whether to connect to the HDFS file system on starting the producer/consumer. If <code>false</code> 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 <code>false</code> allows your application to startup, and not block for up till 15 minutes.

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`, in 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 enough reliable. So, for the moment, it's only possible to create new files. The Camel HDFS endpoint tries to solve this problem in this way:

- If the split strategy option has been defined, the actual file name will become a directory name and a `<file name>/seg0` will be initially created.
- Every time a splitting condition is met a new file is created with name `<original file name>/segN` where N is 1, 2, 3, etc.

The `splitStrategy` option is defined as a string with the following syntax:

`splitStrategy=<ST>:<value>,<ST>:<value>,*`

where `<ST>` can be:

- BYTES a new file is created, and the old is closed when the number of written bytes is more than `<value>`
- MESSAGES a new file is created, and the old is closed when the number of written messages is more than `<value>`
- IDLE a new file is created, and the old is closed when no writing happened in the last `<value>` milliseconds

for example:

```
-----
```

it means: a new file is created either when it has been idle for more than 1 second or if more than 5 bytes have been written. So, running `hadoop fs -ls /tmp/simple-file` you'll find the following files `seg0`, `seg1`, `seg2`, etc

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 explicit 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 is varios strategy that 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.

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.



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.

URI format

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&option=value&...`

Options

Name	Default Value	Description
entityType	entityClassName	Is the provided <i>entityClassName</i> from the URI.
consumeDelete	true	Option for <i>HibernateConsumer</i> only. Specifies whether or not the entity is deleted after it is consumed.
consumeLockEntity	true	Option for <i>HibernateConsumer</i> only. Specifies whether or not to use exclusive locking of each entity while processing the results from the pooling.
flushOnSend	true	Option for <i>HibernateProducer</i> only. Flushes the <i>EntityManager</i> after the entity bean has been persisted.
maximumResults	-1	Option for <i>HibernateConsumer</i> only. Set the maximum number of results to retrieve on the Query.
consumer.delay	500	Option for <i>HibernateConsumer</i> only. Delay in millis between each poll.
consumer.initialDelay	1000	Option for <i>HibernateConsumer</i> only. Millis before polling starts.
consumer.userFixedDelay	false	Option for <i>HibernateConsumer</i> only. Set to <i>true</i> to use fixed delay between polls, otherwise fixed rate is used. See <i>ScheduledExecutorService</i> in JDK for details.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [Hibernate Example](#)

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
- Agnostic data format using either plain String objects or HAPI HL7 model objects.

- Type Converter from/to HAPI and String
- HL7 DataFormat using HAPI library
- Even more ease-of-use as it's integrated well with the camel-mina (**Camel 2.11:** camel-mina2) component.

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

HL7 MLLP protocol

HL7 is often used with the HL7 MLLP protocol that is a text based TCP socket based protocol. This component ships with a Mina Codec that conforms to the MLLP protocol so you can easily expose a HL7 listener that accepts HL7 requests over the TCP transport.

To expose a HL7 listener service we reuse the existing mina/mina2 component where we just use the `HL7MLLPCodec` as codec.

The HL7 MLLP codec has the following options:

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 (is a charset name) to use for the codec. If not provided, Camel will use the JVM default Charset.
convertLFtoCR	true (Camel 2.11: false)	Will convert <code>\n</code> to <code>\r</code> (0x0d, 13 decimal) as HL7 stipulates <code>\r</code> as segment terminators. The HAPI library requires the use of <code>\r</code> .
validate	true	Whether HAPI Parser should validate or not.
parser	ca.uhn.hl7v2.parser.PipeParser	Camel 2.11: To use a custom parser. Must be of type <code>ca.uhn.hl7v2.parser.Parser</code> .

Exposing a HL7 listener

In our Spring XML file, we configure an endpoint to listen for HL7 requests using TCP:

Notice that we use TCP on `localhost` on port 8888. We use **sync=true** to indicate that this listener is synchronous and therefore will return a HL7 response to the caller. Then we setup mina to use our HL7 codec with **codec=#hl7codec**. Notice that `hl7codec` is just a Spring bean ID, so we could have named it `mygreatcodecforhl7` or whatever. The codec is also set up in the Spring XML file:

Above we also configure the charset encoding to use (`iso-8859-1`).

The endpoint **hl7listener** can then be used in a route as a consumer, as this Java DSL example illustrates:

This is a very simple route that will listen for HL7 and route it to a service named **patientLookupService** that is also a Spring bean ID we have configured in the Spring XML as:

Another powerful feature of Camel is that we can have our business logic in POJO classes that is not tied to Camel as shown here:

Notice that this class uses just imports from the HAPI library and **not** from Camel.

HL7 Model using java.lang.String

The HL7MLLP codec uses plain `String` as its data format. Camel uses its Type Converter to convert to/from strings to the HAPI HL7 model objects. However, you can use plain `String` objects if you prefer, for instance if you wish to parse the data yourself.

See samples for such an example.

HL7v2 Model using HAPI

The HL7v2 model uses Java objects from the HAPI library. Using this library, we can encode and decode from the EDI format (ER7) that is mostly used with HL7v2. With this model you can code with Java objects instead of the EDI based HL7 format that can be hard for humans to read and understand.

The sample below is a request to lookup a patient with the patient ID 0101701234.

Using the HL7 model we can work with the data as a `ca.uhn.hl7v2.model.Message` object.

To retrieve the patient ID in the message above, you can do this in Java code:

If you know the message type in advance, you can be more type-safe:

Camel has built-in type converters, so when this operation is invoked:

Camel will convert the received HL7 data from `String` to `Message`. This is powerful when combined with the HL7 listener, then you as the end-user don't have to work with `byte[]`, `String` or any other simple object formats. You can just use the HAPI HL7v2 model objects.

HL7 DataFormat

The HL7 component ships with a HL7 data format that can be used to format between `String` and HL7 model objects.

- `marshal` = from `Message` to byte stream (can be used when returning as response 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:

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:

Here we unmarshal the byte stream into a HAPI Message object that is passed to our patient lookup service.

Notice there is a shorthand syntax in Camel for well-known data formats that is commonly used.

Then you don't need to create an instance of the `HL7DataFormat` object:

Message Headers

The **unmarshal** operation adds these MSH fields as headers on the Camel message:

Key	MSH field	Example
CamelHL7SendingApplication	MSH-3	MYSERVER
CamelHL7SendingFacility	MSH-4	MYSERVERAPP
CamelHL7ReceivingApplication	MSH-5	MYCLIENT
CamelHL7ReceivingFacility	MSH-6	MYCLIENTAPP
CamelHL7Timestamp	MSH-7	20071231235900
CamelHL7Security	MSH-8	null
CamelHL7MessageType	MSH-9-1	ADT
CamelHL7TriggerEvent	MSH-9-2	A01
CamelHL7MessageControl	MSH-10	1234
CamelHL7ProcessingId	MSH-11	P
CamelHL7VersionId	MSH-12	2.4

All headers are `String` types. If a header value is missing, its value is `null`.

Options

The HL7 Data Format supports the following options:

Option	Default	Description
validate	true	Whether the HAPI Parser should validate using the default validation rules. Camel 2.11: better use the <code>parser</code> option and initialize the parser with the desired <code>HAPI ValidationContext</code>
parser	<code>ca.uhn.hl7v2.parser.GenericParser</code>	Camel 2.11: To use a custom parser. Must be of type <code>ca.uhn.hl7v2.parser.Parser</code> . Note that <code>GenericParser</code> also allows to parse XML-encoded HL7v2 messages.

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 since Version 0.6 has been split into a base library and several structure libraries, one for each HL7v2 message version:

- v2.1 structures library



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.



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()`).

- 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 a application works with HL7v2 message versions 2.4 and 2.5 then the following dependencies must be added:

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.

Terser language (Camel 2.11)

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:

HL7 Validation predicate (Camel 2.11)

Often it is preferable to parse a HL7v2 message and validate it against a `HAPI ValidationContext` in a separate step afterwards.

Sample:

HL7 Acknowledgement expression (Camel 2.11)

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:

More Samples

In the following example we send a HL7 request to a HL7 listener and retrieves a response. We use plain `String` types in this example:

In the next sample, we want to route HL7 requests from our HL7 listener to our business logic. We have our business logic in a plain POJO that we have registered in the registry as `hl7service` = for instance using Spring and letting the bean id = `hl7service`.

Our business logic is a plain POJO only using the HAPI library so we have these operations defined:

Then we set up the Camel routes using the `RouteBuilder` as follows:

Notice that we use the HL7 `DataFormat` to enrich our Camel Message with the MSH fields preconfigured on the Camel Message. This lets us much more easily define our routes using the fluent builders.

If we do not use the HL7 `DataFormat`, then we do not gains these headers and we must resort to a different technique for computing the MSH trigger event (= what kind of HL7 message it is). This is a big advantage of the HL7 `DataFormat` over the plain HL7 type converters.

Sample using plain String objects

In this sample we use plain `String` objects as the data format, that we send, process and receive. As the sample is part of a unit test, there is some code for assertions, but you should be able to understand what happens. First we send the plain string, `Hello World`, to the `HL7MLLPCodec` and receive the response as a plain string, `Bye World`.

Here we process the incoming data as plain `String` and send the response also as plain `String`:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

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:

```
.....
```

URI format

```
.....
```

Will by default use port 80 for HTTP and 443 for HTTPS.

Examples

Call the url with the body using POST and return response as out message. If body is null call URL using GET and return response as out message

Java DSL	Spring DSL
----------	------------

.....
-------	-------

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

.....

URI parameters can either be set directly on the endpoint URI or as a header

Java DSL

.....

Set the HTTP request method to POST

Java DSL	Spring DSL
----------	------------

.....
-------	-------



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

HttpEndpoint Options

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 the <code>* throwExceptionOnFailure*</code> to be false to let the <code>HttpProducer</code> send all the fault response back. Camel 2.3: If the option is true, <code>HttpProducer</code> and <code>CamelServlet</code> will skip the gzip processing if the content-encoding is "gzip".
disableStreamCache	false	<code>DefaultHttpBinding</code> will copy the request input stream into a stream cache and put it into message body if this option is false to support read it twice, otherwise <code>DefaultHttpBinding</code> will set the request input stream direct into the message body.
httpBindingRef	null	Deprecated and will be removed in Camel 3.0: Reference to a <code>org.apache.camel.component.http.HttpBinding</code> in the Registry. Use the <code>httpBinding</code> option instead.
httpBinding	null	Camel 2.3: Reference to a <code>org.apache.camel.component.http.HttpBinding</code> in the Registry.
httpClientConfigurerRef	null	Deprecated and will be removed in Camel 3.0: Reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry. Use the <code>httpClientConfigurer</code> option instead.
httpClientConfigurer	null	Camel 2.3: Reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry.
httpClient.XXX	null	Setting options on the <code>HttpClientParams</code> . For instance <code>httpClient.soTimeout=5000</code> will set the <code>SO_TIMEOUT</code> to 5 seconds.
clientConnectionManager	null	To use a custom <code>org.apache.http.conn.ClientConnectionManager</code> .
transferException	false	Camel 2.6: If enabled and an <code>Exchange</code> failed processing on the consumer side, and if the caused <code>Exception</code> was send back serialized in the response as a <code>application/x-java-serialized-object</code> content type (for example using Jetty or <code>SERVLET</code> Camel components). On the producer side the exception will be deserialized and thrown as is, instead of the <code>HttpOperationFailedException</code> . The caused exception is required to be serialized.
headerFilterStrategy	null	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> .
urlRewrite	null	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 <code>UrlRewrite</code> and <code>How to use Camel as a HTTP proxy between a client and server</code> .

Authentication and Proxy

The following authentication options can also be set on the `HttpEndpoint`:

Name	Default Value	Description
authMethod	null	Authentication method, either as Basic, Digest or NTLM.
authMethodPriority	null	Priority of authentication methods. Is a list separated with comma. For example: Basic, Digest to exclude NTLM.
authUsername	null	Username for authentication
authPassword	null	Password for authentication
authDomain	null	Domain for 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 Basic, Digest or NTLM.
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.

HttpComponent Options

Name	Default Value	Description
<code>httpBinding</code>	null	To use a custom <code>org.apache.camel.component.http.HttpBinding</code> .
<code>httpClientConfigurer</code>	null	To use a custom <code>org.apache.camel.component.http.HttpClientConfigurer</code> .
<code>httpConnectionManager</code>	null	To use a custom <code>org.apache.commons.httpclient.HttpConnectionManager</code> .
<code>httpConfiguration</code>	null	To use a custom <code>org.apache.camel.component.http.HttpConfiguration</code>

`HttpConfiguration` contains all the options listed in the table above under the section *HttpConfiguration - Setting Authentication and Proxy*.

Message Headers

Name	Type	Description
<code>Exchange.HTTP_URI</code>	String	URI to call. Will override existing URI set directly on the endpoint.
<code>Exchange.HTTP_METHOD</code>	String	HTTP Method / Verb to use (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE)
<code>Exchange.HTTP_PATH</code>	String	Request URI's path, the header will be used to build the request URI with the <code>HTTP_URI</code> . Camel 2.3.0: If the path is start with <code>"/"</code> , http producer will try to find the relative path based on the <code>Exchange.HTTP_BASE_URI</code> header or the <code>exchange.getFromEndpoint().getEndpointUri()</code> ;
<code>Exchange.HTTP_QUERY</code>	String	URI parameters. Will override existing URI parameters set directly on the endpoint.
<code>Exchange.HTTP_RESPONSE_CODE</code>	int	The HTTP response code from the external server. Is 200 for OK.
<code>Exchange.HTTP_CHARACTER_ENCODING</code>	String	Character encoding
<code>Exchange.CONTENT_TYPE</code>	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> .
<code>Exchange.CONTENT_ENCODING</code>	String	The HTTP content encoding. Is set on both the IN and OUT message to provide a content encoding, such as <code>gzip</code> .
<code>Exchange.HTTP_SERVLET_REQUEST</code>	<code>HttpServletRequest</code>	The <code>HttpServletRequest</code> object.
<code>Exchange.HTTP_SERVLET_RESPONSE</code>	<code>HttpServletResponse</code>	The <code>HttpServletResponse</code> object.
<code>Exchange.HTTP_PROTOCOL_VERSION</code>	String	Camel 2.5: You can set the http protocol version with this header, eg. <code>"HTTP/1.0"</code> . If you didn't specify the header, <code>HttpProducer</code> will use the default value <code>"HTTP/1.1"</code>

The header name 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.

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 get access to HttpServletRequest and HttpServletResponse

You can get access to these two using the Camel type converter system using

```
-----
```



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.

Using client timeout - SO_TIMEOUT

See the unit test in this link

MORE EXAMPLES

Configuring a Proxy

Java DSL

```
.....
[ ]
```

There is also support for proxy authentication via the `proxyUsername` and `proxyPassword` options.

Using proxy settings outside of URI

Java DSL

Spring DSL

```
.....
[ ] [ ]
```

Options on Endpoint will override options on the context.

Configuring charset

If you are using `POST` to send data you can configure the `charset`

```
[ ]
```

Sample with scheduled poll

The sample polls the Google homepage every 10 seconds and write the page to the file `message.html`:

```
[ ]
```

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 `HttpProducer.HTTP_RESPONSE_CODE`.

Using `throwExceptionOnFailure=false` to get any response back

In the route below we want to route a message that we enrich with data returned from a remote HTTP call. As we want any response from the remote server, we set the `throwExceptionOnFailure` option to `false` so we get any response in the `AggregationStrategy`. As the code is based on a unit test that simulates a HTTP status code 404, there is some assertion code etc.

Disabling Cookies

To disable cookies you can set the HTTP Client to ignore cookies by adding this URI option: `httpClient.cookiePolicy=ignoreCookies`

Advanced Usage

If you need more control over the HTTP producer you should use the `HttpComponent` where you can set various classes to give you custom behavior.

Setting `MaxConnectionsPerHost`

The HTTP Component has a `org.apache.commons.httpclient.HttpConnectionManager` where you can configure various global configuration for the given component.

By global, we mean that any endpoint the component creates has the same shared `HttpConnectionManager`. So, if we want to set a different value for the max connection per host, we need to define it on the HTTP component and **not** on the endpoint URI that we usually use. So here comes:

First, we define the `http` component in Spring XML. Yes, we use the same scheme name, `http`, because otherwise 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.

And then we can just use it as we normally do in our routes:

Using preemptive authentication

An end user reported that he had problem with authenticating with HTTPS. The problem was eventually resolved when he discovered the HTTPS server did not return a HTTP code 401 Authorization Required. The solution was to set the following 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

As of 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.SSLContextParametersSecureProtocolSocketFactory` 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.

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:

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 route builder class you can hook it up like so:

If you are doing this using the Spring DSL, you can specify your `HttpClientConfigurer` using the URI. For example:

As long as you implement the `HttpClientConfigurer` and configure your keystore and truststore as described above, it will work fine.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [Jetty](#)

IBATIS

The **ibatis:** component allows you to query, poll, insert, update and delete data in a relational database using Apache iBATIS.
Maven users will need to add the following dependency to their `pom.xml` for this component:

URI format

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

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

This component will by default load the iBatis `SqlMapConfig` file from the root of the classpath and expected named as `SqlMapConfig.xml`.
It uses Spring resource loading so you can define it using `classpath`, `file` or `http` as prefix to load resources with those schemes.
In Camel 2.2 you can configure this on the `iBatisComponent` with the `setSqlMapConfig(String)` method.

Options

Option	Type	Default	Description
<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 comma.
<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.



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.

consumer.routeEmptyResultSet	boolean	false	Sets whether empty result set should be routed or not. By default, empty result sets are not routed.
statementType	StatementType	null	Mandatory to specify for IbatisProducer to control which iBatis SqlMapClient method to invoke. The enum values are: QueryForObject, QueryForList, Insert, Update, Delete.
maxMessagesPerPoll	int	0	An integer to define a maximum messages to gather per poll. By default, no maximum is set. Can be used to set a limit of e.g. 1000 to avoid when starting up the server that there are thousands of files. Set a value of 0 or negative to disabled it.
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
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:

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:

```
[...]
```

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:

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:

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`:

And the same for `UPDATE`, where we can send an `Account` object as IN body to iBatis:

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:

And the iBatis SQL map file used:

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.

And the statements in the sqlmap file:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [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:

```
<code></code>
```

URI format

```
<code></code>
```

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:MyNick@irc.server.org#channel or irc:irc.server.org#channel?nickname=MyUser	null
username	The IRC server user name.	irc:MyUser@irc.server.org#channel or irc:irc.server.org#channel?username=MyUser	Same as nickname.
password	The IRC server password.	password=somepass	None
realname	The IRC user's actual name.	realname=MyName	None
colors	Whether or not the server supports color codes.	true, false	true
onReply	Whether or not to handle general responses to commands or informational messages.	true, false	false
onNick	Handle nickname change events.	true, false	true
onQuit	Handle user quit events.	true, false	true
onJoin	Handle user join events.	true, false	true
onKick	Handle kick events.	true, false	true
onMode	Handle mode change events.	true, false	true
onPart	Handle user part events.	true, false	true
onTopic	Handle topic change events.	true, false	true
onPrivmsg	Handle message events.	true, false	true
trustManager	The trust manager used to verify the SSL server's certificate.	trustManager=#referenceToTrustManagerBean	The default trust manager, which accepts all certificates, will be used.

keys	Camel 2.2: Comma separated list of IRC channel keys. Important to be listed in same order as channels. When joining multiple channels with only some needing keys just insert an empty value for that channel.	irc:MyNick@irc.server.org/#channel?keys=chankey	null
sslContextParameters	Camel 2.9: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry. This reference overrides any configured <code>SSLContextParameters</code> at the component level. See Using the JSSE Configuration Utility. Note that this setting overrides the <code>trustManager</code> 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

Spring DSL based configuration of endpoint

Using the legacy basic configuration options

You can also connect to an SSL enabled IRC server, as follows:

By default, the IRC transport uses `SSLDefaultTrustManager`. If you need to provide your own custom trust manager, use the `trustManager` parameter as follows:

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.

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:

```
<code></code>
```

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:

```
<code></code>
```

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.

```
<code></code>
```

Which outputs the following result

```
<code></code>
```

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:

```
<code></code>
```

Which outputs the following result:

```
<code></code>
```

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)`

```
<code></code>
```

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.

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.

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

Then start the application, such as running the start script.

When the application is up and running you can unset the environment

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:

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)`



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.

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.

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.

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:

URI format

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

Options

Name	Default Value	Description
spaceName	null	Specifies the JavaSpace name.
verb	take	Specifies the verb for getting JavaSpace entries. The values can be: take or read.
transactional	false	If true, sending and receiving entries is performed within a transaction.
transactionalTimeout	Long.MAX_VALUE	Specifies the transaction timeout.
concurrentConsumers	1	Specifies the number of concurrent consumers getting entries from the JavaSpace.
templateId	null	If present, this option specifies the Spring bean ID of the template to use for reading/taking entries.

Examples

Sending and Receiving Entries

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:

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](#)

JBIFCOMPONENT

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.

The following code:

```
Automaticallv 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:

The messages sent by this producer endpoint are sent to the already deployed JBI endpoint.

URI format

The separator that should be used in the endpoint URL is:

- / (forward slash), if `serviceNamespace` starts with `http://`, or
- : (colon), if `serviceNamespace` starts with `urn:foo:bar`.

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

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:

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

Examples

URI options

Name	Default value	Description
------	---------------	-------------



See below for information about how to use `StreamSource` types from ServiceMix in Camel.

<code>mep</code>	MEP of the Camel Exchange	Allows users to override the MEP set on the Exchange object. Valid values for this option are <code>in-only</code> , <code>in-out</code> , <code>robust-in-out</code> and <code>in-optional-out</code> .
<code>operation</code>	Value of the <code>jbi.operation</code> header property	Specifies the JBI operation for the <code>MessageExchange</code> . If no value is supplied, the JBI binding will use the value of the <code>jbi.operation</code> header property.
<code>serialization</code>	<code>basic</code>	Default value (<code>basic</code>) will check if headers are serializable by looking at the type, setting this option to <code>strict</code> will detect objects that can not be serialized although they implement the <code>Serializable</code> interface. Set to <code>nocheck</code> to disable this check altogether, note that this should only be used for in-memory transports like <code>SEDAFlow</code> , otherwise you can expect to get <code>NotSerializableException</code> thrown at runtime.
<code>convertException</code>	<code>false</code>	<code>false</code> : send any exceptions thrown from the Camel route back unmodified <code>true</code> : convert all exceptions to a <code>JBI FaultException</code> (can be used to avoid non-serializable exceptions or to implement generic error handling)

Examples

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.

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:

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:

URI format



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>	<code>CamelJcrInsert</code> or <code>CamelJcrGetById</code> operation to use
<code>CamelJcrNodeName</code>	<code>null</code>	Used to determine the node name to use.

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 properties of the IN message are transformed to `JCR Value` instances and added to the new node and the node's UUID is returned in the OUT message.
- If the operation is `CamelJcrGetById`: A new node is retrieved from the repository using the message body as node identifier.

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>uuids</code>	<code>null</code>	Only events whose associated parent node has one of the identifiers in the comma separated uuid list will be received.
<code>nodeTypeNames</code>	<code>null</code>	Only events whose associated parent node has one of the node types (or a subtype of one of the node types) in this list will be received.
<code>noLocal</code>	<code>false</code>	If <code>noLocal</code> is true, then events generated by the session through which the listener was registered are ignored. Otherwise, they are not ignored.
<code>sessionLiveCheckInterval</code>	<code>60000</code>	Interval in milliseconds to wait before each session live checking.
<code>sessionLiveCheckIntervalOnStart</code>	<code>3000</code>	Interval in milliseconds to wait before the first session live checking.

Example

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

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.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

JDBC COMPONENT

The **jdbc** component enables you to access databases through JDBC, where SQL queries and operations are sent in the message body. This component uses the standard JDBC API, unlike the SQL Component component, which uses `spring-jdbc`.

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

URI format

This component only supports producer endpoints.

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

Options

Name	Default Value	Description
<code>readSize</code>	0	The default maximum number of rows that value is 0.



This component can only be used to define producer endpoints, which means that you cannot use the JDBC component in a `from()` statement.



This component can not be used as a Transactional Client. If you need transaction support in your route, you should use the SQL component instead.

<code>statement.<xxx></code>	<code>null</code>	Camel 2.1: Sets additional options on the statement behind the scenes to execute the queries. For detailed documentation, see the java
<code>useJDBC4ColumnNameAndLabelSemantics</code>	<code>true</code>	Camel 2.2: Sets whether to use JDBC 4/3 semantics for this option to turn it <code>false</code> in case you have legacy data. This only applies when using SQL SE as identifier, name as given_
<code>resetAutoCommit</code>	<code>true</code>	Camel 2.9: Camel will set the <code>autoCommit</code> flag to <code>false</code> after executed the statement and commit the change after executed the statement. At the end of the connection at the end, if the <code>resetAutoCommit</code> flag is set to <code>true</code> , support to reset the <code>autoCommit</code> flag, you can set it to <code>false</code> , and Camel will not try to reset the <code>autoCommit</code>
<code>allowNamedParameters</code>	<code>true</code>	Camel 2.12: Whether to allow using named parameters
<code>prepareStatementStrategy</code>	<code>Ë</code>	Camel 2.12: Allows to plugin to use a custom <code>PreparedStatement</code> strategy. The default is <code>org.apache.camel.component.jdbc.DefaultPreparedStatementStrategy</code> to control preparation of the query and prepare
<code>useHeadersAsParameters</code>	<code>false</code>	Camel 2.12: Set this option to <code>true</code> to use headers as named parameters. This allows to define headers with the dynamic values for the query

Result

The result is returned in the OUT body as an `ArrayList<HashMap<String, Object>>`. The `List` object contains the list of rows and the `Map` objects contain each row with the `String` key as the column name.

Note: This component fetches `ResultSetMetaData` to be able to return the column name as the key in the `Map`.

Message Headers

Header	Description
CamelJdbcRowCount	If the query is a <code>SELECT</code> , query the row count is returned in this OUT header.
CamelJdbcUpdateCount	If the query is an <code>UPDATE</code> , query the update count is returned in this OUT header.
CamelGeneratedKeysRows	Camel 2.10: Rows that contains the generated kets.
CamelGeneratedKeysRowCount	Camel 2.10: The number of rows in the header that contains generated keys.
CamelJdbcColumnNames	Camel 2.11.1: The column names from the <code>ResultSet</code> as a <code>java.util.Set</code> type.
CamelJdbcParametes	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 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:

```
-----
```

You can also store the header values in a `java.util.Map` and store the map on the headers with the key `CamelJdbcParameters`.



Using generated keys does not work with together with named parameters.

Samples

In the following example, we fetch the rows from the customer table.

First we register our datasource in the Camel registry as `testdb`:

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:

Or you can create a `DataSource` in Spring like this:

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:

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:

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:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [SQL](#)

JETTY COMPONENT

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 a http client which mean you can also use it with Camel as a producer. Maven users will need to add the following dependency to their `pom.xml` for this component:



Stream

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 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 re-read multiple times.

URI format

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

Options

Name	Default Value	Description
<code>sessionSupport</code>	<code>false</code>	Specifies whether to enable the session manager on the server side of Jetty.
<code>httpClient.XXX</code>	<code>null</code>	Configuration of Jetty's <code>HttpClient</code> . 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.
<code>httpClient</code>	<code>null</code>	To use a shared <code>org.eclipse.jetty.client.HttpClient</code> for all producers created by this endpoint. This option should only be used in special circumstances.
<code>httpClientMinThreads</code>	<code>null</code>	Camel 2.11: Producer only: To set a value for minimum number of threads in <code>HttpClient</code> thread pool. This setting override any setting configured on component level. Notice that both a min and max size must be configured.
<code>httpClientMaxThreads</code>	<code>null</code>	Camel 2.11: Producer only: To set a value for maximum number of threads in <code>HttpClient</code> thread pool. This setting override any setting configured on component level. Notice that both a min and max size must be configured.
<code>httpBindingRef</code>	<code>null</code>	Reference to an <code>org.apache.camel.component.http.HttpBinding</code> in the Registry. <code>HttpBinding</code> can be used to customize how a response should be written for the consumer.
<code>jettyHttpBindingRef</code>	<code>null</code>	Camel 2.6.0+: Reference to an <code>org.apache.camel.component.jetty.JettyHttpBinding</code> in the Registry. <code>JettyHttpBinding</code> can be used to customize how a response should be written for the producer.
<code>matchOnUriPrefix</code>	<code>false</code>	Whether or not the <code>CamelServlet</code> 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 .
<code>handlers</code>	<code>null</code>	Specifies a comma-delimited set of <code>org.mortbay.jetty.Handler</code> instances in your Registry (such as your Spring <code>ApplicationContext</code>). These handlers are added to the Jetty servlet context (for example, to add security). Important: 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.
<code>chunked</code>	<code>true</code>	Camel 2.2: If this option is false Jetty servlet will disable the HTTP streaming and set the content-length header on the response
<code>enableJmx</code>	<code>false</code>	Camel 2.3: If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details
<code>disableStreamCache</code>	<code>false</code>	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 <code>true</code> when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. Default <code>HttpBinding</code> will copy the request input stream into a stream cache and put it into message body if this option is <code>false</code> 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.
<code>throwExceptionOnFailure</code>	<code>true</code>	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.

transferException	false	Camel 2.6: If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a <code>application/x-java-serialized-object</code> content type. On the producer side the exception will be deserialized and thrown as is, instead of the <code>HttpOperationFailedException</code> . The caused exception is required to be serialized.
bridgeEndpoint	false	Camel 2.1: 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 the throwExceptionOnFailure to be false to let the <code>HttpProducer</code> send all the fault response back. Camel 2.3: If the option is true, <code>HttpProducer</code> and <code>CamelServlet</code> will skip the gzip processing if the content-encoding is "gzip". Also consider setting disableStreamCache to true to optimize when bridging.
enableMultipartFilter	true	Camel 2.5: Whether <code>Jetty org.eclipse.jetty.servlets.MultipartFilter</code> is enabled or not. You should set this value to false when bridging endpoints, to ensure multipart requests is proxied/bridged as well.
multipartFilterRef	null	Camel 2.6: Allows using a custom multipart filter. Note: setting <code>multipartFilterRef</code> forces the value of <code>enableMultipartFilter</code> to true.
filtersRef	null	Camel 2.9: Allows using a custom filters which is putted into a list and can be find in the Registry
continuationTimeout	null	Camel 2.6: Allows to set a timeout in millis when using <code>Jetty</code> as consumer (server). By default <code>Jetty</code> uses 30000. You can use a value of <code><= 0</code> to never expire. If a timeout occurs then the request will be expired and <code>Jetty</code> will return back a http error 503 to the client. This option is only in use when using <code>Jetty</code> with the <code>Asynchronous Routing Engine</code> .
useContinuation	true	Camel 2.6: Whether or not to use <code>Jetty</code> continuations for the <code>Jetty Server</code> .
sslContextParametersRef	null	Camel 2.8: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry.É This reference overrides any configured <code>SSLContextParameters</code> at the component level.É See Using the JSSE Configuration Utility.
traceEnabled	false	Specifies whether to enable HTTP TRACE for this <code>Jetty</code> consumer. By default TRACE is turned off.
headerFilterStrategy	null	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>HttpJettyEndpoint</code> .
urlRewrite	null	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 <code>UrlRewrite</code> and <code>How to use Camel as a HTTP proxy between a client and server</code> .
responseBufferSize	null	Camel 2.12: To use a custom buffer size on the <code>javax.servlet.ServletResponse</code> .

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 turn on or turn off the chuched 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.

Starting with Camel 2.2.0, you can get the `request.parameter` from the message header not only from `Get Method`, but also other HTTP method.

Usage

The `Jetty` component 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:

Name	Default Value	Description
enableJmx	false	Camel 2.3: If this option is true, <code>Jetty JMX</code> support will be enabled for this endpoint. See <code>Jetty JMX</code> support for more details.

<code>sslKeyPassword</code>	<code>null</code>	Consumer only: The password for the keystore when using SSL.
<code>sslPassword</code>	<code>null</code>	Consumer only: The password when using SSL.
<code>sslKeystore</code>	<code>null</code>	Consumer only: The path to the keystore.
<code>minThreads</code>	<code>null</code>	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.
<code>maxThreads</code>	<code>null</code>	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.
<code>threadPool</code>	<code>null</code>	Camel 2.5 Consumer only: To use a custom thread pool for the server. This option should only be used in special circumstances.
<code>sslSocketConnectors</code>	<code>null</code>	Camel 2.3 Consumer only: A map which contains per port number specific SSL connectors. See section <i>SSL support</i> for more details.
<code>socketConnectors</code>	<code>null</code>	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 <i>SSL support</i> for more details.
<code>sslSocketConnectorProperties</code>	<code>null</code>	Camel 2.5 Consumer only: A map which contains general SSL connector properties. See section <i>SSL support</i> for more details.
<code>socketConnectorProperties</code>	<code>null</code>	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 <i>SSL support</i> for more details.
<code>httpClient</code>	<code>null</code>	Deprecated: Producer only: To use a custom <code>HttpClient</code> with the jetty producer. This option is removed from Camel 2.11 onwards, instead you can set the option on the endpoint instead.
<code>httpClientMinThreads</code>	<code>null</code>	Producer only: To set a value for minimum number of threads in <code>HttpClient</code> thread pool. Notice that both a min and max size must be configured.
<code>httpClientMaxThreads</code>	<code>null</code>	Producer only: To set a value for maximum number of threads in <code>HttpClient</code> thread pool. Notice that both a min and max size must be configured.
<code>httpClientThreadPool</code>	<code>null</code>	Deprecated: Producer only: To use a custom thread pool for the client. This option is removed from Camel 2.11 onwards.
<code>sslContextParameters</code>	<code>null</code>	Camel 2.8: To configure a custom SSL/TLS configuration options at the component level. See Using the JSSE Configuration Utility for more details.
<code>requestBufferSize</code>	<code>null</code>	Camel 2.11.2: Allows to configure a custom value of the request buffer size on the Jetty connectors.
<code>requestHeaderSize</code>	<code>null</code>	Camel 2.11.2: Allows to configure a custom value of the request header size on the Jetty connectors.
<code>responseBufferSize</code>	<code>null</code>	Camel 2.11.2: Allows to configure a custom value of the response buffer size on the Jetty connectors.
<code>responseHeaderSize</code>	<code>null</code>	Camel 2.11.2: Allows to configure a custom value of the response header size on the Jetty connectors.

Producer Example

The following is a basic example of how to send an HTTP request to an existing HTTP endpoint.

in Java DSL

```

// ...

```

or in Spring XML

```

<!-- ... -->

```

Consumer Example

In this sample we define a route that exposes a HTTP service at `http://localhost:8080/myapp/myservice`:

```

// ...

```

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.

```

// ...

```



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.

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`.

So if a client sends the HTTP request, `http://serverUri?one=hello`, the Jetty component will copy the HTTP request parameter, `one` to the exchange's `in.header`. We can then use the `simple` language to route exchanges that contain this header to a specific endpoint and all others to another. If we used a language more powerful than Simple--such as `EL` or `OGNL`--we could also test for the parameter value and do routing based on the header value as well.

Session Support

The session support option, `sessionSupport`, can be used to enable a `HttpSession` object and access the session object while processing the exchange. For example, the following route enables sessions:

The `myCode` Processor can be instantiated by a Spring bean element:

Where the processor implementation can access the `HttpSession` as follows:

SSL Support (HTTPS)

Using the JSSE Configuration Utility

As of Camel 2.8, the 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

Spring DSL based configuration of endpoint

Configuring Jetty Directly

Jetty provides SSL support out of the box. To enable Jetty to run in SSL mode, simply format the URI with the `https://` prefix---for example:

Jetty also needs to know where to load your keystore from and what passwords to use in order to load the correct SSL certificate. Set the following JVM System Properties:

until Camel 2.2

- `jetty.ssl.keystore` specifies the location of the Java keystore file, which contains the Jetty server's own X.509 certificate in a *key entry*. A key entry stores the X.509 certificate (effectively, the *public key*) and also its associated private key.
- `jetty.ssl.password` the store password, which is required to access the keystore file (this is the same password that is supplied to the `keystore` command's `-storepass` option).
- `jetty.ssl.keypassword` the key password, which is used to access the certificate's key entry in the keystore (this is the same password that is supplied to the `keystore` command's `-keypass` option).

from Camel 2.3 onwards

- `org.eclipse.jetty.ssl.keystore` specifies the location of the Java keystore file, which contains the Jetty server's own X.509 certificate in a *key entry*. A key entry stores the X.509 certificate (effectively, the *public key*) and also its associated private key.
- `org.eclipse.jetty.ssl.password` the store password, which is required to access the keystore file (this is the same password that is supplied to the `keystore` command's `-storepass` option).
- `org.eclipse.jetty.ssl.keypassword` the key password, which is used to access the certificate's key entry in the keystore (this is the same password that is supplied to the `keystore` command's `-keypass` option).

For details of how to configure SSL on a Jetty endpoint, read the following documentation at the Jetty Site: <http://docs.codehaus.org/display/JETTY/How+to+configure+SSL>

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

Camel 2.3, 2.4

*From Camel 2.5 we switch to use `SslSelectChannelConnector` *

The value you use as keys in the above map is the port you configure Jetty to listen on.

Configuring general SSL properties

Available as of Camel 2.5

Instead of a per port number specific SSL socket connector (as shown above) you can now configure general properties which applies for all SSL socket connectors (which is not explicit configured as above with the port number as entry).

How to obtain reference to the X509Certificate

Jetty stores a reference to the certificate in the `HttpServletRequest` which you can access from code as follows:

Configuring general HTTP properties

Available as of Camel 2.5

Instead of a per port number specific HTTP socket connector (as shown above) you can now configure general properties which applies for all HTTP socket connectors (which is not explicit configured as above with the port number as entry).

Default behavior for returning HTTP status codes

The default behavior of HTTP status codes is defined by the `org.apache.camel.component.http.DefaultHttpBinding` class, which handles how a response is written and also sets the HTTP status code.

If the exchange was processed successfully, the 200 HTTP status code is returned. If the exchange failed with an exception, the 500 HTTP status code is returned, and the `stacktrace` is returned in the body. If you want to specify which HTTP status code to return, set the code in the `HttpProducer.HTTP_RESPONSE_CODE` header of the OUT message.

Customizing HttpBinding

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:

We can then create an instance of our binding and register it in the Spring registry as follows:

And then we can reference this binding when we define the 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:

And from Camel 2.3 onwards you can configure a list of Jetty handlers as follows:

You can then define the endpoint as:

If you need more handlers, set the `handlers` option equal to a comma-separated list of bean IDs.

How to return a custom HTTP 500 reply message

You may want to return a custom reply message when something goes wrong, instead of the default reply message Camel Jetty replies with.

You could use a custom `HttpBinding` to be in control of the message mapping, but often it may be easier to use Camel's Exception Clause to construct the custom reply message. For example as show here, where we return `Dude something went wrong` with HTTP error code 500:

Multi-part Form support

From Camel 2.3.0, camel-jetty support to multipart form post out of box. The submitted form-data are mapped into the message header. Camel-jetty creates an attachment for each uploaded file. The file name is mapped to the name of the attachment. The content type is set as the content type of the attachment file name. You can find the example here.

Listing 1. 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 that JMX must be enabled within the Camel context in order to enable JMX support in this component as the component provides Jetty with a reference to the MBeanServer registered with the Camel context. Because the camel-jetty component caches and reuses Jetty resources for a given protocol/host/port pairing, this configuration option will only be evaluated during the creation of the first endpoint to use a protocol/host/port pairing. For example, given two routes created from the following XML fragments, JMX support would remain enabled for all endpoints listening on "https://0.0.0.0".

The camel-jetty component also provides for direct configuration of the Jetty MBeanContainer. Jetty creates MBean names dynamically. If you are running another instance of Jetty outside of the Camel context and sharing the same MBeanServer between the instances, you can provide both instances with a reference to the same MBeanContainer in order to avoid name collisions when registering Jetty MBeans.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [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:

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

URI format

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

Example	Description
---------	-------------

<code>rng:foo/bar.rng</code>	References the XML file foo/bar.rng on the classpath
<code>rnc: http://foo.com/bar.rnc</code>	References the RelaxNG Compact Syntax file from the URL, <code>http://foo.com/bar.rnc</code>

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

Options

Option	Default	Description
<code>useDom</code>	<code>false</code>	Specifies whether DOMSource/DOMResult or SaxSource/SaxResult should be used by the validator.

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).

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

JMS COMPONENT

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

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

URI format



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.

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:

```
-----
```

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

```
-----
```

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

```
-----
```

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

If you are consuming messages and using transactions (`transacted=true`) then the default settings for cache level can 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 onwards, the default setting for `cacheLevelName` is `CACHE_AUTO`. This default auto detects the mode and sets the cache level accordingly to:

- `CACHE_CONSUMER` = if `transacted=false`
- `CACHE_NONE` = if `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 as follows:

- 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

Options

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

The options are divided into two tables, the first one with the most common options used. The latter contains the rest.

Most commonly used options

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>	<code>1</code>	Specifies the default number of concurrent consumers. From Camel 2.10.3 onwards this option can also be used when doing request/reply over JMS. See also the <code>maxMessagesPerTask</code> option to control dynamic scaling up/down of threads.
<code>disableReplyTo</code>	<code>false</code>	If <code>true</code> , 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>	<code>1</code>	Specifies the maximum number of concurrent consumers. From Camel 2.10.3 onwards this option can also be used when doing request/reply over JMS. See also the <code>maxMessagesPerTask</code> option to control dynamic scaling up/down of threads.
<code>maxMessagesPerTask</code>	<code>-1</code>	The number of messages per task. <code>-1</code> is unlimited. If you use a range for concurrent consumers (eg <code>min < max</code>), then this option can be used to set a value to eg <code>100</code> to control how fast the consumers will shrink when less work is required.
<code>preserveMessageQos</code>	<code>false</code>	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 three headers are considered <code>JMSPriority</code> , <code>JMSDeliveryMode</code> , and <code>JMSExpiration</code> . You can provide all or only some 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>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: <code>Temporary</code> , <code>Shared</code> , or <code>Exclusive</code> . By default Camel will use temporary 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 ones. See further below for more details, and especially the notes about the implications if running in a clustered environment, and the fact that <code>Shared</code> 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 <code>InOut</code> Exchange Pattern (in milliseconds). The default is 20 seconds. 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 such as <code>=</code> as <code>%3D</code> Before Camel 2.3.0 , we don't support this option 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.



Mapping to Spring JMS

Many of these properties map to properties on Spring JMS, which Camel uses for sending and receiving messages. So you can get more information about these properties by consulting the relevant Spring documentation.

transacted	false	Specifies whether to use transacted mode for sending/receiving messages using the InOnly 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 onwards also the JMS producers is tested as well.

All the other options

Option	Default Value	Description
acceptMessagesWhileStopping	false	Specifies whether the consumer accept messages while it is stopping. You may consider enabling this option, if you start and stop JMS routes at runtime, while there are still messages enqueued on the queue. If this option is <code>false</code> , and you stop the JMS route, then messages may be rejected, and the JMS broker would have to attempt redeliveries, which yet again may be rejected, and eventually the message may be moved at a dead letter queue on the JMS broker. To avoid this its recommended to enable this option.
acknowledgementModeName	AUTO_ACKNOWLEDGE	The JMS acknowledgement name, which is one of: <code>SESSION_TRANSACTED</code> , <code>CLIENT_ACKNOWLEDGE</code> , <code>AUTO_ACKNOWLEDGE</code> , <code>DUPS_OK_ACKNOWLEDGE</code>
acknowledgementMode	-1	The JMS acknowledgement mode defined as an Integer. Allows you to set vendor-specific extensions to the acknowledgement mode. For the regular modes, it is preferable to use the <code>acknowledgementModeName</code> instead.
allowNullBody	true	Camel 2.9.3/2.10.1: Whether to allow sending messages with no body. If this option is <code>false</code> and the message body is null, then an <code>JMSException</code> is thrown.
alwaysCopyMessage	false	If <code>true</code> , Camel will always make a JMS message copy of the message when it is passed to the producer for sending. Copying the message is needed in some situations, such as when a <code>replyToDestinationSelectorName</code> is set (incidentally, Camel will set the <code>alwaysCopyMessage</code> option to <code>true</code> , if a <code>replyToDestinationSelectorName</code> is set)
asyncConsumer	false	Camel 2.9: Whether the <code>JmsConsumer</code> processes the Exchange asynchronously. If enabled then the <code>JmsConsumer</code> may pickup the next message from the JMS queue, while the previous message is being processed asynchronously (by the Asynchronous Routing Engine). This means that messages may be processed not 100% strictly in order. If disabled (as default) then the Exchange is fully processed before the <code>JmsConsumer</code> will pickup the next message from the JMS queue. Note if <code>transacted</code> has been enabled, then <code>asyncConsumer=true</code> does not run asynchronously, as transactions must be executed synchronously (Camel 3.0 may support async transactions).
asyncStartListener	false	Camel 2.10: Whether to startup the <code>JmsConsumer</code> message listener asynchronously, when starting a route. For example if a <code>JmsConsumer</code> cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to <code>true</code> , you will let routes startup, while the <code>JmsConsumer</code> connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at <code>WARN</code> level, and the consumer will not be able to receive messages; You can then restart the route to retry.
asyncStopListener	false	Camel 2.10: Whether to stop the <code>JmsConsumer</code> message listener asynchronously, when stopping a route.
autoStartup	true	Specifies whether the consumer container should auto-startup.
cacheLevelName	CACHE_AUTO (Camel >= 2.8.0) CACHE_CONSUMER (Camel <= 2.7.1)	Sets the cache level by name for the underlying JMS resources. Possible values are: <code>CACHE_AUTO</code> , <code>CACHE_CONNECTION</code> , <code>CACHE_CONSUMER</code> , <code>CACHE_NONE</code> , and <code>CACHE_SESSION</code> . The default setting for Camel 2.8 and newer is <code>CACHE_AUTO</code> . For Camel 2.7.1 and older the default is <code>CACHE_CONSUMER</code> . See the Spring documentation and Transactions Cache Levels for more information.
cacheLevel	É	Sets the cache level by ID for the underlying JMS resources. See <code>cacheLevelName</code> option for more details.

consumerType	Default	The consumer type to use, which can be one of: Simple, Default, or Custom. The consumer type determines which Spring JMS listener to use. Default will use <code>org.springframework.jms.listener.DefaultMessageListenerContainer</code> , Simple will use <code>org.springframework.jms.listener.SimpleMessageListenerContainer</code> . When Custom is specified, the <code>MessageListenerContainerFactory</code> defined by the <code>messageListenerContainerFactoryRef</code> option will determine what <code>org.springframework.jms.listener.AbstractMessageListenerContainer</code> to use (new option in Camel 2.10.2 onwards). This option was temporary removed in Camel 2.7 and 2.8. But has been added back from Camel 2.9 onwards.
connectionFactory	null	The default JMS connection factory to use for the <code>listenerConnectionFactory</code> and <code>templateConnectionFactory</code> , if neither is specified.
defaultTaskExecutorType	(see description)	Camel 2.10.4: Specifies what default <code>TaskExecutor</code> type to use in the <code>DefaultMessageListenerContainer</code> , for both consumer endpoints and the <code>ReplyTo</code> consumer of producer endpoints. Possible values: <code>SimpleAsync</code> (uses Spring's <code>SimpleAsyncTaskExecutor</code>) or <code>ThreadPool</code> (uses Spring's <code>ThreadPoolTaskExecutor</code> with optimal values - cached threadpool-like). If not set, it defaults to the previous behaviour, which uses a cached thread pool for consumer endpoints and <code>SimpleAsync</code> for reply consumers. The use of <code>ThreadPool</code> is recommended to reduce "thread trash" in elastic configurations with dynamically increasing and decreasing concurrent consumers.
deliveryPersistent	true	Specifies whether persistent delivery is used by default.
destination	null	Specifies the JMS Destination object to use on this endpoint.
destinationName	null	Specifies the JMS destination name to use on this endpoint.
destinationResolver	null	A pluggable <code>org.springframework.jms.support.destination.DestinationResolver</code> that allows you to use your own resolver (for example, to lookup the real destination in a JNDI registry).
disableTimeToLive	false	Camel 2.8: Use this option to force disabling time to live. For example when you do request/reply over JMS, then Camel will by default use the <code>requestTimeout</code> value as time to live on the message being sent. The problem is that the sender and receiver systems have to have their clocks synchronized, so they are in sync. This is not always so easy to archive. So you can use <code>disableTimeToLive=true</code> to not set a time to live value on the sent message. Then the message will not expire on the receiver system. See below in section <i>About time to live</i> for more details.
eagerLoadingOfProperties	false	Enables eager loading of JMS properties as soon as a message is received, which is generally inefficient, because the JMS properties might not be required. But this feature can sometimes catch early any issues with the underlying JMS provider and the use of JMS properties. This feature can also be used for testing purposes, to ensure JMS properties can be understood and handled correctly.
exceptionListener	null	Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.
errorHandler	null	Camel 2.8.2, 2.9: Specifies a <code>org.springframework.util.ErrorHandler</code> to be invoked in case of any uncaught exceptions thrown while processing a <code>Message</code> . By default these exceptions will be logged at the WARN level, if no <code>errorHandler</code> has been configured. From Camel 2.9.1: onwards you can configure logging level and whether stack traces should be logged using the below two options. This makes it much easier to configure, than having to code a custom <code>ErrorHandler</code> .
errorHandlerLoggingLevel	WARN	Camel 2.9.1: Allows to configure the default <code>errorHandler</code> logging level for logging uncaught exceptions.
errorHandlerLogStackTrace	true	Camel 2.9.1: Allows to control whether stacktraces should be logged or not, by the default <code>ErrorHandler</code> .
explicitQosEnabled	false	Set if the <code>deliveryMode</code> , <code>priority</code> or <code>timeToLive</code> qualities of service should be used when sending messages. This option is based on Spring's <code>JmsTemplate</code> . The <code>deliveryMode</code> , <code>priority</code> and <code>timeToLive</code> options are applied to the current endpoint. This contrasts with the <code>preserveMessageQos</code> option, which operates at message granularity, reading QoS properties exclusively from the Camel In message headers.
exposeListenerSession	true	Specifies whether the listener session should be exposed when consuming messages.
forceSendOriginalMessage	false	Camel 2.7: When using <code>mapJmsMessage=false</code> Camel will create a new JMS message to send to a new JMS destination if you touch the headers (get or set) during the route. Set this option to <code>true</code> to force Camel to send the original JMS message that was received.
idleTaskExecutionLimit	1	Specifies the limit for idle executions of a receive task, not having received any message within its execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in the case of dynamic scheduling; see the <code>maxConcurrentConsumers</code> setting). There is additional doc available from Spring.
idleConsumerLimit	1	Camel 2.8.2, 2.9: Specify the limit for the number of consumers that are allowed to be idle at any given time.
includeSentJMSMessageID	false	Camel 2.10.3: Only applicable when sending to JMS destination using <code>InOnly</code> (eg fire and forget). Enabling this option will enrich the Camel Exchange with the actual <code>JMSMessageID</code> that was used by the JMS client when the message was sent to the JMS destination.

includeAllJMSXProperties	false	Camel 2.11.2/2.12: Whether to include all JMSxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSAppID, and JMSUserID etc. Note: If you are using a custom headerFilterStrategy then this option does not apply.
jmsMessageType	null	Allows you to force the use of a specific javax.jms.Message implementation for sending JMS messages. Possible values are: Bytes, Map, Object, Stream, Text. By default, Camel would determine which JMS message type to use from the In body type. This option allows you to specify it.
jmsKeyFormatStrategy	default	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides two implementations out of the box: default and passthrough. The default strategy will safely marshal dots and hyphens (. and -). The passthrough strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the org.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the # notation.
jmsOperations	null	Allows you to use your own implementation of the org.springframework.jms.core.JmsOperations interface. Camel uses JmsTemplate as default. Can be used for testing purpose, but not used much as stated in the spring API docs.
lazyCreateTransactionManager	true	If true, Camel will create a JmsTransactionManager, if there is no transactionManager injected when option transacted=true.
listenerConnectionFactory	null	The JMS connection factory used for consuming messages.
mapJmsMessage	true	Specifies whether Camel should auto map the received JMS message to an appropriate payload type, such as javax.jms.TextMessage to a String etc. See section about how mapping works below for more details.
maximumBrowseSize	-1	Limits the number of messages fetched at most, when browsing endpoints using Browse or JMX API.
messageConverter	null	To use a custom Spring org.springframework.jms.support.converter.MessageConverter so you can be 100% in control how to map to/from a javax.jms.Message.
messageIdEnabled	true	When sending, specifies whether message IDs should be added.
messageListenerContainerFactoryRef	null	Camel 2.10.2: Registry ID of the MessageListenerContainerFactory used to determine what org.springframework.jms.listener.AbstractMessageListenerContainer to use to consume messages. Setting this will automatically set consumerType to Custom.
messageTimestampEnabled	true	Specifies whether timestamps should be enabled by default on sending messages.
password	null	The password for the connector factory.
priority	4	Values greater than 1 specify the message priority when sending (where 0 is the lowest priority and 9 is the highest). The explicitQosEnabled option must also be enabled in order for this option to have any effect.
pubSubNoLocal	false	Specifies whether to inhibit the delivery of messages published by its own connection.
receiveTimeout	None	The timeout for receiving messages (in milliseconds).
recoveryInterval	5000	Specifies the interval between recovery attempts, i.e. when a connection is being refreshed, in milliseconds. The default is 5000 ms, that is, 5 seconds.
replyToCacheLevelName	CACHE_CONSUMER	Camel 2.9.1: Sets the cache level by name for the reply consumer when doing request/reply over JMS. This option only applies when using fixed reply queues (not temporary). Camel will by default use: CACHE_CONSUMER for exclusive or shared w/ replyToSelectorName. And CACHE_SESSION for shared without replyToSelectorName. Some JMS brokers such as IBM WebSphere may require to set the replyToCacheLevelName=CACHE_NONE to work. Note: If using temporary queues then CACHE_NONE is not allowed, and you must use a higher value such as CACHE_CONSUMER or CACHE_SESSION.
replyToDestinationSelectorName	null	Sets the JMS Selector using the fixed name to be used so you can filter out your own replies from the others when using a shared queue (that is, if you are not using a temporary reply queue).
replyToDeliveryPersistent	true	Specifies whether to use persistent delivery by default for replies.
requestTimeoutCheckerInterval	1000	Camel 2.9.2: Configures how often Camel should check for timed out Exchanges when doing request/reply over JMS. By default Camel checks once per second. But if you must react faster when a timeout occurs, then you can lower this interval, to check more frequently. The timeout is determined by the option requestTimeout.
subscriptionDurable	false	@deprecated: Enabled by default, if you specify a durableSubscriberName and a clientId.
taskExecutor	null	Allows you to specify a custom task executor for consuming messages.
taskExecutorSpring2	null	Camel 2.6: To use when using Spring 2.x with Camel. Allows you to specify a custom task executor for consuming messages.
templateConnectionFactory	null	The JMS connection factory used for sending messages.

transactedInOut	false	@deprecated: Specifies whether to use transacted mode for sending messages using the InOut Exchange Pattern. Applies only to producer endpoints. See section Enabling Transacted Consumption 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), if using transacted mode.
transferException	false	If enabled and you are using Request Reply messaging (InOut) and an Exchange failed on the consumer side, then the caused Exception will be send back in response as a <code>javax.jms.ObjectMessage</code> . If the client is Camel, the returned Exception is rethrown. This allows you to use Camel JMS as a bridge in your routing - for example, using persistent queues to enable robust routing. Notice that if you also have transferExchange enabled, this option takes precedence. The caught exception is required to be serializable. The original Exception on the consumer side can be wrapped in an outer exception such as <code>org.apache.camel.RuntimeCamelException</code> when returned to the producer.
transferExchange	false	You can transfer the exchange over the wire instead of just the body and headers. The following fields are transferred: In body, Out body, Fault body, In headers, Out headers, Fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at <code>WARN</code> level. You must enable this option on both the producer and consumer side, so Camel knows the payloads is an Exchange and not a regular payload.
username	null	The username for the connector factory.
useMessageIDAsCorrelationID	false	Specifies whether <code>JMSMessageID</code> should always be used as <code>JMSCorrelationID</code> for InOut messages.
useVersion102	false	@deprecated (removed from Camel 2.5 onwards): Specifies whether the old JMS API should 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
String	<code>javax.jms.TextMessage</code>	£
<code>org.w3c.dom.Node</code>	<code>javax.jms.TextMessage</code>	The DOM will be converted to String.
Map	<code>javax.jms.MapMessage</code>	£
<code>java.io.Serializable</code>	<code>javax.jms.ObjectMessage</code>	£
<code>byte[]</code>	<code>javax.jms.BytesMessage</code>	£
<code>java.io.File</code>	<code>javax.jms.BytesMessage</code>	£
<code>java.io.Reader</code>	<code>javax.jms.BytesMessage</code>	£
<code>java.io.InputStream</code>	<code>javax.jms.BytesMessage</code>	£
<code>java.nio.ByteBuffer</code>	<code>javax.jms.BytesMessage</code>	£

When receiving a JMS message, Camel converts the JMS message to the following body type:

JMS Message	Body Type
<code>javax.jms.TextMessage</code>	String
<code>javax.jms.BytesMessage</code>	<code>byte[]</code>
<code>javax.jms.MapMessage</code>	<code>Map<String, Object></code>
<code>javax.jms.ObjectMessage</code>	Object

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:

```
route().to("jms:orderQueue", messageConverter(myCustomMessageConverter()));
```

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:

```
route().pollFrom("file:./data?jmsMessageType=Text").to("jms:orderQueue");
```

You can also specify the message type to use for each message by setting the header with the key `CamelJmsMessageType`. For example:

```
route().pollFrom("file:./data").to("jms:orderQueue", header("CamelJmsMessageType", "Text"));
```

The possible values are defined in the `enum` class, `org.apache.camel.jms.JmsMessageType`.

Message format when sending

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

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

- Keys starting with `JMS` or `JMSX` are reserved.
- `exchange.in.headers` keys must be literals and all be valid Java identifiers (do not use dots in the key name).
- Camel replaces dots & hyphens and the reverse when consuming JMS messages:
 - `.` is replaced by `_DOT_` and the reverse replacement when Camel consumes the message.

– is replaced by `_HYPHEN_` and the reverse replacement when Camel consumes the message.

- See also the option `jmsKeyFormatStrategy`, which allows use of your own custom strategy for formatting keys.

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

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

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

```
.....
```

Message format when receiving

Camel adds the following properties to the `Exchange` when it receives a message:

Property	Type	Description
<code>org.apache.camel.jms.replyDestination</code>	<code>javax.jms.Destination</code>	The reply destination.

Camel adds the following JMS properties to the In message headers when it receives a JMS message:

Header	Type	Description
<code>JMSCorrelationID</code>	<code>String</code>	The JMS correlation ID.
<code>JMSDeliveryMode</code>	<code>int</code>	The JMS delivery mode.
<code>JMSDestination</code>	<code>javax.jms.Destination</code>	The JMS destination.
<code>JMSExpiration</code>	<code>long</code>	The JMS expiration.
<code>JMSMessageID</code>	<code>String</code>	The JMS unique message ID.
<code>JMSPriority</code>	<code>int</code>	The JMS priority (with 0 as the lowest priority and 9 as the highest).
<code>JMSRedelivered</code>	<code>boolean</code>	Is the JMS message redelivered.
<code>JMSReplyTo</code>	<code>javax.jms.Destination</code>	The JMS reply-to destination.
<code>JMSTimestamp</code>	<code>long</code>	The JMS timestamp.
<code>JMSType</code>	<code>String</code>	The JMS type.
<code>JMSXGroupID</code>	<code>String</code>	The JMS group ID.

As all the above information is standard JMS you can check the JMS documentation for further details.

About using Camel to send and receive messages and `JMSReplyTo`

The JMS component is complex and you have to pay close attention to how it works in some cases. So this is a short summary of some of the areas/pitfalls to look for.

When Camel sends a message using its `JMSProducer`, it checks the following conditions:

- The message exchange pattern,
- Whether a `JMSReplyTo` was set in the endpoint or in the message headers,

- All this can be a tad complex to understand and configure to support your use case.

The `JmsProducer` behaves as follows, depending on configuration:

JmsConsumer

The `JmsConsumer` behaves as follows, depending on configuration:

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:

Reuse endpoint and send to different destinations computed 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:

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:

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:

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` takes priority.

Configuring different JMS providers

You can configure your JMS provider in Spring XML as follows:

Basically, you can configure as many JMS component instances as you wish and give them a **unique name using the `id` attribute**. The preceding example configures an `activemq` component. You could do the same to configure `MQSeries`, `TibCo`, `BEA`, `Sonic` and so on.

Once you have a named JMS component, you can then refer to endpoints within that component using URIs. For example for the component name, `activemq`, you can then refer to destinations using the URI format, `activemq:[queue:|topic:]destinationName`. You can use the same approach for all other JMS providers.

This works by the `SpringCamelContext` lazily fetching components from the spring context for the scheme name you use for Endpoint URIs and having the Component resolve the endpoint URIs.

Using JNDI to find the ConnectionFactory

If you are using a J2EE container, you might need to look up JNDI to find the JMS `ConnectionFactory` rather than use the usual `<bean>` mechanism in Spring. You can do this using Spring's factory bean or the new Spring XML namespace. For example:

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:

You can configure this option in one of the following ways:

- On the `JmsComponent`,
- On the endpoint URI or,
- By invoking `setConcurrentConsumers()` directly on the `JmsEndpoint`.

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 summaries the options.

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 <code>Shared</code> 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 <code>Exclusive</code> 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.
<code>concurrentConsumers</code>	Fast	Yes	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. Notice: That using <code>Shared</code> reply queues may not work as well with concurrent listeners, so use this option with care.
<code>maxConcurrentConsumers</code>	Fast	Yes	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. Notice: That using <code>Shared</code> reply queues may not work as well with concurrent listeners, so use this option with care.

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 automatic setup a consumer which listen on the reply queue, so you should **not** do anything.

This consumer is a `Spring DefaultMessageListenerContainer` which listen 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:

In this route we instruct Camel to route replies asynchronously using a thread pool with 5 threads.

From Camel 2.10.3 onwards 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:

Request-reply over JMS and 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.

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 1000 millis. So to make it faster you can set it to 250 millis to pull 4 times per second as shown:

Notice this will cause the Camel to send pull requests to the message broker more frequent, and thus require more network traffic.

It is generally recommended to use temporary queues if possible.

Request-reply over JMS and 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 selectors 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** onwards 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:

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:

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 with Camel then Camel uses a timeout on the sender side, which is default 20 seconds 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 send. So that requires the clocks to 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** onwards. 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.

When you do fire and forget (InOut) over JMS with Camel then Camel by default does **not** set any time to live value on the message. You can configure a value by 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

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 = a Transaction Manager` - typically the `JmsTransactionManager`

See the Transactional Client EIP pattern for further details.

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:

```
exchange.getProperty(Exchange.REPLY_TO_DESTINATION);
```

And then later use it to send a reply using regular JMS or Camel.

```
context.send(destination, message);
```

A different solution to sending a reply is to provide the `replyDestination` 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. For example:

```
context.send(endpointUri, message, exchange);
```

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:

```
context.requestBody(endpointUri, message, InOut, timeout);
```

Samples

JMS is used in many examples for other components as well. But we provide a few samples below to get started.



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 for Request Reply(InOut MEP), you **must** set `transactedInOut=true`.

Receiving from JMS

In the following sample we configure a route that receives JMS messages and routes the message to a POJO:

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:

Sending to a JMS

In the sample below we poll a file folder and send the file content to a JMS topic. As we want the content of the file as a `TextMessage` instead of a `BytesMessage`, we need to convert the body to a `String`:

Using Annotations

Camel also has annotations so you can use POJO Consuming and POJO Producing.

Spring DSL sample

The preceding examples use the Java DSL. Camel also supports Spring XML DSL. Here is the big spender sample using Spring DSL:

```
-----
```

Other samples

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`. The demo below illustrates this:

```
-----
```

Then you can consume from the JMS queue and analyze the problem:

```
-----
```

Using JMS as a Dead Letter Channel storing error only

You can use JMS to store the cause error message or to store a custom body, which you can initialize yourself. The following example uses the Message Translator EIP to do a transformation on the failed exchange before it is moved to the JMS dead letter queue:

```
-----
```

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 its InOnly or InOut messaging. However there can be times where you want to send an

InOnly message but keeping the JMSReplyTo header. To do so you have to instruct Camel to keep it, otherwise the JMSReplyTo header will be dropped.

For example to send an InOnly message to the foo queue, but with a JMSReplyTo with bar queue you can do as follows:

Notice 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:

Some versions of WMQ won't accept this option on the destination name and you will get an exception like:

```
com.ibm.msg.client.jms.DetailedJMSException: JMSSC0005: The specified value
'MY_QUEUE?targetClient=1' is not allowed for 'XMSC_DESTINATION_NAME'
```

A workaround is to use a custom `DestinationResolver`:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [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:

URI Format

The component can connect to the local platform mbean server with the following URI:

A remote mbean server url can be provided following the initial JMX scheme like so:

You can append query options to the URI in the following format,
?options=value&option2=value&...

URI Options

Property	Required	Default	Description
format	Ê	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	Ê	Ê	Credentials for making a remote connection.
password	Ê	Ê	Credentials for making a remote connection.
objectDomain	yes	Ê	The domain for the mbean you're connecting to.
objectName	Ê	Ê	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	Ê	Ê	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	Ê	Ê	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	Ê	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 ð where 'x' is the configured <code>reconnectDelay</code> .
reconnectOnConnectionFailure	Ê	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 <code>reconnectDelay</code> .
reconnectDelay	Ê	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 URI must always have the `objectDomain` property. In addition, the URI must contain either `objectName` or one or more properties that start with "key."

Domain with Name property

When the `objectName` property is provided, the following constructor is used to build the `ObjectName?` for the mbean:

The key value in the above will be "name" and the value will be the value of the `objectName` property.

Domain with Hashtable

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

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:

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

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
granualityPeriod	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 notificaton
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.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [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:

```
<code></code>
```

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 does not contain one of the previous listed types, put a Message Translator in front of the endpoint to perform the necessary conversion first.

Consuming from the endpoint

Consuming messages from a JPA consumer endpoint removes (or updates) entity beans in the database. This allows you to use a database table as a logical queue: consumers take messages from the queue and then delete/update them to logically remove them from the queue.

If you do not wish to delete the entity bean when it has been processed (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).

URI format

```
<code></code>
```

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	entityClassName	Overrides the <i>entityClassName</i> from the URI.
persistenceUnit	camel	The JPA persistence unit used by default.
consumeDelete	true	JPA consumer only: If <i>true</i> , the entity is deleted after it is consumed; if <i>false</i> , 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 <i>EntityManager</i> after the entity bean has been persisted.
maximumResults	-1	JPA consumer only: Set the maximum number of results to retrieve on the Query.
transactionManager	null	This option is Registry based which requires the # notation so that the given <i>transactionManager</i> being specified can be looked up properly, e.g. <i>transactionManager=#myTransactionManager</i> . It specifies the transaction manager to use. If none provided, Camel will use a <i>JpaTransactionManager</i> 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 <i>true</i> to use fixed delay between polls, otherwise fixed rate is used. See <i>ScheduledExecutorService</i> 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	É	JPA consumer only: To use a custom query when consuming data.
consumer.namedQuery	É	JPA consumer only: To use a named query when consuming data.
consumer.nativeQuery	É	JPA consumer only: To use a custom native query when consuming data. You may want to use the option <i>consumer.resultClass</i> also when using native queries.
consumer.parameters	É	Camel 2.12: JPA consumer only: the parameters map which will be used for building the query. The parameters is an instance of Map which key is String and value is Object.
consumer.resultClass	É	Camel 2.7: JPA consumer only: Defines the type of the returned payload (we will call <i>entityManager.createNativeQuery(nativeQuery, resultClass)</i> instead of <i>entityManager.createNativeQuery(nativeQuery)</i>). 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 (<i>false</i>) 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 <i>javax.persistence.LockModeType</i> .
usePersist	false	Camel 2.5: JPA producer only: Indicates to use <i>entityManager.persist(entity)</i> instead of <i>entityManager.merge(entity)</i> . Note: <i>entityManager.persist(entity)</i> doesn't work for detached entities (where the <i>EntityManager</i> has to execute an UPDATE instead of an INSERT query)!

Message Headers

Camel adds the following message headers to the exchange:

Header	Type	Description
CamelJpaTemplate	JpaTemplate	The <i>JpaTemplate</i> object that is used to access the entity bean. You need this object in some situations, for instance in a type converter or when you are doing some custom processing.

Configuring EntityManagerFactory

Its 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:

```

<!-- Example configuration for JPA component using a specific EntityManagerFactory -->

```


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

Its strongly advised to configure the `TransactionManager` instance used by the JPA component. If failed to do so each `JpaEndpoint` will auto create their own instance of `TransactionManager` which most often is not what you want.

For example, you can instantiate a JPA component that references the `myTransactionManager` transaction manager, as follows:

In **Camel 2.3** the `JpaComponent` will auto lookup the `TransactionManager` from the Registry which means you do not need to configure this on the `JpaComponent` as shown above. You only need to do so if there is ambiguity, in which case Camel will log a WARN.

Using a consumer with a named query

For consuming only selected entities, you can use the `consumer.namedQuery` URI query option. First, you have to define the named query in the JPA Entity class:

After that you can define a consumer uri like this one:

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:

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:

If you use the native query option, 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:

Second we have to setup a `org.springframework.orm.jpa.JpaTemplate` which is used by the

```
org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository:
```

Afterwards we can configure our

```
org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository:
```

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

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [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:

URI format

To call remote program (**Camel 2.7**)

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
------	---------------	-------------



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:

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:

Then it will all become green again 😊

ccsid	default system CCSID	Specifies the CCSID to use for the connection with the AS/400 system.
format	text	Specifies the data format for sending messages valid options are: text (represented by String) and binary (represented by byte[])
consumer.delay	500	Delay in milliseconds between each poll.
consumer.initialDelay	1000	Milliseconds before polling starts.
consumer.userFixedDelay	false	true to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
guiAvailable	false	Camel 2.8: Specifies whether AS/400 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.

For the remote program call (Camel 2.7)

Name	Default value	Description
outputFieldsIdx	Ê	Specifies which fields (program parameters) are output parameters.
fieldsLength	Ê	Specifies the fields (program parameters) length as in the AS/400 program definition.
format	text	Camel 2.10: Specifies the data format for sending messages valid options are: text (represented by String) and binary (represented by byte[])
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 explicit 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.

```
-----
```

Remote program call example (Camel 2.7)

In the snippet below, the data Exchange sent to the `direct:work` endpoint will contain three string that will be used as the arguments for the program `ÔcomputeÔ` in the library `ÔassetsÔ`. This program will write the output values in the 2nd and 3rd parameters. All the parameters will be sent to the `direct:play` endpoint.

```
-----
```

Writing to keyed data queues

```
-----
```

Reading from keyed data queues

```
-----
```

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

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

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 <code>simple</code> , <code>groovy</code> , <code>javascript</code> 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 <code>false</code> the script is executed but the result of the script is discarded.
contentCache	true	boolean	Camel 2.9: Whether to cache the script if loaded from a resource. Note: from Camel 2.10.3 a cached script can be forced to reload at runtime via JMX using the <code>clearContentCache</code> operation.

Message Headers

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

Header	Description
<code>CamelLanguageScript</code>	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:

In case you want to convert the message body type you can do this as well:

You can also use the Groovy language, such as this example where the input message will be multiplied with 2:

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.

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:

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 **Camel 2.11** onwards you can refer to the resource similar to the other Languages in Camel by prefixing with `"resource:"` as shown below:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [Languages](#)
- [Routing Slip](#)
- [Dynamic Router](#)

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:

```
<code></code>
```

URI format

```
<code></code>
```

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 <code>object</code> , <code>onelevel</code> , or <code>subtree</code> .
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 <code>LdapContext</code> (subclass of <code>DirContext</code>) has to be passed in as <code>LdapServerBean</code> (otherwise an exception is thrown)
returnedAttributes	depends on LDAP Server (could be all or none)	Camel 2.6: Comma-separated list of attributes that should be set in each entry of the result

Result

The result is returned in the Out body as a
`ArrayList<javax.naming.directory.SearchResult>` object.

DirContext

The URI, `ldap:ldapserver`, references a Spring bean with the ID, `ldapserver`. The `ldapserver` bean may be defined as follows:

```
<code></code>
```

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.

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:

Binding using credentials

A Camel end user donated this sample code he used to bind to the ldap server using credentials.

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

Where **loggingCategory** is the name of the logging category to use. You can append query options to the URI in the following format, `?option=value&option=value&...`

For example, a log endpoint typically specifies the logging level using the `level` option, as follows:

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.

Options

Option	Default	Type	Description
level	INFO	String	Logging level to use. Possible values: ERROR, WARN, INFO, DEBUG, TRACE, OFF
marker	null	String	Camel 2.9: An optional Marker name to use.
groupSize	null	Integer	An integer that specifies a group size for throughput logging.
groupInterval	null	Integer	If specified will group message stats by this time interval (in millis)
groupDelay	0	Integer	Set the initial delay for stats (in millis)
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

note: `groupDelay` and `groupActiveOnly` are only applicable when using `groupInterval`

Formatting

The log formats the execution of exchanges to log lines.
By default, the log uses `LogFormatter` to format the log output, where `LogFormatter` has the following options:

Option	Default	Description
showAll	false	Quick option for turning all options on. (multiline, maxChars has to be manually set if to be used)
showExchangeId	false	Show the unique exchange ID.
showExchangePattern	true	Shows the Message Exchange Pattern (or MEP for short).
showProperties	false	Show the exchange properties.
showHeaders	false	Show the In message headers.
showBodyType	true	Show the In body java type.
showBody	true	Show the In body.
showOut	false	If the exchange has an Out message, show the Out message.
showException	false	If the exchange has an exception, show the exception message (no stack trace).
showCaughtException	false	If the exchange has a caught exception, show the exception message (no stack trace). A caught exception is stored as a property on the exchange (using the key <code>Exchange.EXCEPTION_CAUGHT</code>) and for instance a <code>doCatch</code> can catch exceptions. See <code>Try Catch Finally</code> .
showStackTrace	false	Show the stack trace, if an exchange has an exception. Only effective if one of <code>showAll</code> , <code>showException</code> or <code>showCaughtException</code> are enabled.
showFiles	false	Camel 2.9: Whether Camel should show file bodies or not (eg such as <code>java.io.File</code>).
showFuture	false	Whether Camel should show <code>java.util.concurrent.Future</code> bodies or not. If enabled Camel could potentially wait until the <code>Future</code> task is done. Will by default not wait.
showStreams	false	Camel 2.8: Whether Camel should show stream bodies or not (eg such as <code>java.io.InputStream</code>). Beware if you enable this option then you may not be able later to access the message body as the stream have already been read by this logger. To remedy this you will have to use <code>Stream</code> caching.
multiline	false	If true, each piece of information is logged on a new line.
maxChars	£	Limits the number of characters logged per line. The default value is 10000 from Camel 2.9 onwards.



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](#).



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.setLogLevel(LoggingLevel.INFO);
```

Regular logger sample

In the route below we log the incoming orders at `DEBUG` level before the order is processed:

```
route().log("Incoming order", true).process(processOrder());
```

Or using Spring XML to define the route:

```
<route id="orderRoute">
  <log message="Incoming order" level="DEBUG" />
  <process ref="processOrder" />
</route>
```

Regular logger with formatter sample

In the route below we log the incoming orders at `INFO` level before the order is processed.

```
route().log("Incoming order", true, "%s", order.getId()).process(processOrder());
```

Throughput logger with groupSize sample

In the route below we log the throughput of the incoming orders at `DEBUG` level grouped by 10 messages.

```
route().logThroughput("Incoming order", true, 10).process(processOrder());
```

Throughput logger with groupInterval sample

This route will result in message stats logged every 10s, with an initial 60s delay and stats should be displayed even if there isn't any message traffic.

```
route().logThroughput("Incoming order", true, 10, 60, 10).process(processOrder());
```

The following will be logged:

```
2016-01-01 10:00:00.000 [main] DEBUG org.apache.camel.Logging: Incoming order: 10 messages
```

Full customization of the logging 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:

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 either of two ways:

Explicitly instantiating the LogComponent in your Registry:

Convention over configuration:

Simply by registering a bean with the name `logFormatter`; the Log Component is intelligent enough to pick it up automatically.

NOTE: the `ExchangeFormatter` gets applied to **all Log endpoints within that Camel Context**. If you need different `ExchangeFormatters` for different endpoints, 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** onwards 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 "logForamtter" as prototype scoped so its not shared if you have different parameters, eg:

And then we can have Camel routes using the log uri with different options:

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:

```
<code></code>
```

URI format

```
<code></code>
```

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

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

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

Example 2: Loading properties into the JNDI registry in the Camel Context

Example 2: Performing searches using a Query Producer

Example 3: Performing searches using a Query Processor

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:

URI format

Mail endpoints can have one of the following URI formats (for the protocols, SMTP, POP3, or IMAP, respectively):

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:

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):

Alternatively, it is possible to specify both the user name and the password as query options:

For example:

Default ports

Default port numbers are supported. If the port number is omitted, Camel determines the port number to use based on the protocol.

Protocol	Default Port Number
SMTP	25
SMTPS	465
POP3	110
POP3S	995
IMAP	143
IMAPS	993



Geronimo mail .jar

We have discovered that the `geronimo mail .jar` (v1.6) has a bug when polling mails with attachments. It cannot correctly identify the `Content-Type`. So, if you attach a `.jpeg` file to a mail and you poll it, the `Content-Type` is resolved as `text/plain` and not as `image/jpeg`. For that reason, we have added an `org.apache.camel.component.ContentTypeResolver` SPI interface which enables you to provide your own implementation and fix this bug by returning the correct Mime type based on the file name. So if the file name ends with `jpeg/jpg`, you can return `image/jpeg`.

You can set your custom resolver on the `MailComponent` instance or on the `MailEndpoint` instance.



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.

Options

Property	Default	Description
<code>host</code>	<code>É</code>	The host name or IP address to connect to.
<code>port</code>	See DefaultPorts	The TCP port number to connect on.
<code>username</code>	<code>É</code>	The user name on the email server.
<code>password</code>	<code>null</code>	The password on the email server.
<code>ignoreUriScheme</code>	<code>false</code>	If <code>false</code> , Camel uses the scheme to determine the transport protocol (POP, IMAP, SMTP etc.)
<code>defaultEncoding</code>	<code>null</code>	The default encoding to use for Mime Messages.
<code>contentType</code>	<code>text/plain</code>	The mail message content type. Use <code>text/html</code> for HTML mails.
<code>folderName</code>	<code>INBOX</code>	The folder to poll.
<code>destination</code>	<code>username@host</code>	@deprecated Use the <code>to</code> option instead. The TO recipients (receivers of the email).
<code>to</code>	<code>username@host</code>	The TO recipients (the receivers of the mail). Separate multiple email addresses with a comma.
<code>replyTo</code>	<code>alias@host</code>	As of Camel 2.8.4, 2.9.1+ , the Reply-To recipients (the receivers of the response mail). Separate multiple email addresses with a comma.
<code>CC</code>	<code>null</code>	The CC recipients (the receivers of the mail). Separate multiple email addresses with a comma.

BCC	null	The BCC recipients (the receivers of the mail). Separate multiple email addresses with a comma.
from	camel@localhost	The FROM email address.
subject	Ê	As of Camel 2.3 , the Subject of the message being sent. Note: Setting the subject in the header takes precedence over this option.
delete	false	Deletes the messages after they have been processed. This is done by setting the <code>DELETED</code> flag on the mail message. If <code>false</code> , the <code>SEEN</code> flag is set instead. As of Camel 2.10 you can override this configuration option by setting a header with the key <code>delete</code> to determine if the mail should be deleted or not.
unseen	true	It is possible to configure a consumer endpoint so that it processes only unseen messages (that is, new messages) or all messages. Note that Camel always skips deleted messages. The default option of <code>true</code> will filter to only unseen messages. POP3 does not support the <code>SEEN</code> flag, so this option is not supported in POP3; use IMAP instead. Important: This option is not in use if you also use <code>searchTerm</code> options. Instead if you want to disable unseen when using <code>searchTerm</code> 's then add <code>searchTerm.unseen=false</code> as a term.
copyTo	null	Camel 2.10: Consumer only. After processing a mail message, it can be copied to a mail folder with the given name. You can override this configuration value, with a header with the key <code>copyTo</code> , allowing you to copy messages to folder names configured at runtime.
fetchSize	-1	Sets the maximum number of messages to consume during a poll. This can be used to avoid overloading a mail server, if a mailbox folder contains a lot of messages. Default value of <code>-1</code> means no fetch size and all messages will be consumed. Setting the value to 0 is a special corner case, where Camel will not consume any messages at all.
alternativeBodyHeader	CamelMailAlternativeBody	Specifies the key to an IN message header that contains an alternative email body. For example, if you send emails in text/html format and want to provide an alternative mail body for non-HTML email clients, set the alternative mail body with this key as a header.
debugMode	false	Enable debug mode on the underlying mail framework. The SUN Mail framework logs the debug messages to <code>System.out</code> by default.
connectionTimeout	30000	The connection timeout in milliseconds. Default is 30 seconds.
consumer.initialDelay	1000	Milliseconds before the polling starts.
consumer.delay	60000	Camel will poll the mailbox only once a minute by default to avoid overloading the mail server.
consumer.useFixedDelay	false	Set to <code>true</code> to use a fixed delay between polls, otherwise fixed rate is used. See <code>ScheduledExecutorService</code> in JDK for details.
disconnect	false	Camel 2.8.3/2.9: Whether the consumer should disconnect after polling. If enabled this forces Camel to connect on each poll.
closeFolder	true	Camel 2.10.4: Whether the consumer should close the folder after polling. Setting this option to <code>false</code> and having <code>disconnect=false</code> as well, then the consumer keep the folder open between polls.
mail.XXX	null	Set any additional java mail properties. For instance if you want to set a special property when using POP3 you can now provide the option directly in the URI such as: mail.pop3.forgettopheaders=true. You can set multiple such options, for example: mail.pop3.forgettopheaders=true&mail.mime.encodefilename=true.
mapMailMessage	true	Camel 2.8: Specifies whether Camel should map the received mail message to Camel body/headers. If set to <code>true</code> , the body of the mail message is mapped to the body of the Camel IN message and the mail headers are mapped to IN headers. If this option is set to <code>false</code> then the IN message contains a raw <code>javax.mail.Message</code> . You can retrieve this raw message by calling <code>exchange.getIn().getBody(javax.mail.Message.class)</code> .
maxMessagesPerPoll	0	Specifies the maximum number of messages to gather per poll. By default, no maximum is set. Can be used to set a limit of e.g. 1000 to avoid downloading thousands of files when the server starts up. Set a value of 0 or negative to disable this option.
javaMailSender	null	Specifies a pluggable <code>org.springframework.mail.javamail.JavaMailSender</code> instance in order to use a custom email implementation. If none provided, Camel uses the default <code>org.springframework.mail.javamail.JavaMailSenderImpl</code> .
ignoreUnsupportedCharset	false	Option to let Camel ignore unsupported charset in the local JVM when sending mails. If the charset is unsupported then <code>charset=XXX</code> (where XXX represents the unsupported charset) is removed from the <code>content-type</code> and it relies on the platform default instead.
sslContextParameters	null	Camel 2.10: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry.Ê This reference overrides any configured <code>SSLContextParameters</code> at the component level.Ê See Using the JSSE Configuration Utility.
searchTerm	null	Camel 2.11: Refers to a <code>javax.mail.search.SearchTerm</code> which allows to filter mails based on search criteria such as subject, body, from, sent after a certain date etc. See further below for examples.
searchTerm.xxx	null	Camel 2.11: To configure search terms directly from the endpoint uri, which supports a limited number of terms defined by the <code>org.apache.camel.component.mail.SimpleSearchTerm</code> class. See further below for examples.

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

Spring DSL based configuration of endpoint

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.

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

The same applies for other `MimeMessage` headers such as recipients, so you can use a header property as `To`:

Since Camel 2.11 When using the `MailProducer` to 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.

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:

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:

SUN JavaMail

SUN JavaMail is used under the hood for consuming and producing mails. We encourage end-users to consult these references when using either POP3 or IMAP protocol. Note particularly that POP3 has a much more limited set of features than IMAP.

- SUN POP3 API
- SUN 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`.

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.

In this sample we want to send a mail to multiple recipients:

Sending mail with attachment sample

The mail component supports attachments. In the sample below, we send a mail message containing a plain text message with a logo file attachment.

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.

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:

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:

Instead of logging the mail we use a processor where we can process the mail from java code:

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



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.

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:

```
split("${body.messages}")
```

If you use XML DSL then you need to declare a method call expression in the Splitter as shown below

```
split("${body.messages}", "${body.messages}.splitAttachmentsExpression()")
```

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:

```
camelMailEndpoint.searchTerm("${searchTerm.subjectOrBody} Camel")
```

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.

```
camelMailEndpoint.unseen("${now-24h"}")
```

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:

```
camelMailEndpoint.unseen("${now-24h"}", "${searchTerm.subjectOrBody} Camel")
```

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

You can then refer to this bean, using `#beanId` in your Camel route as shown:

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:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

MINA COMPONENT

Deprecated

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:

URI format

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.



Deprecated

This component is deprecated as the Apache Mina 1.x project is EOL. Instead use MINA2 or Netty instead.

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

Option	Default Value	Description
codec	null	You can refer to a named <code>ProtocolCodecFactory</code> instance in your Registry such as your Spring <code>ApplicationContext</code> , which is then used for the marshalling.
codec	null	You must use the <code>#</code> 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 <code>false</code> , 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: <code>DEFAULT</code> , <code>AUTO</code> , <code>WINDOWS</code> , <code>UNIX</code> or <code>MAC</code> . If none provided, Camel will use <code>DEFAULT</code> . 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 serializable. Camel will exclude any non-serializable objects and log it at <code>WARN</code> level.
minaLogger	false	You can enable the Apache MINA logging filter. Apache MINA uses <code>slf4j</code> logging at <code>INFO</code> level to log all input and output.
filters	null	You can set a list of Mina <code>IoFilters</code> to register. The <code>filters</code> 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.

<code>allowDefaultCodec</code>	<code>true</code>	The mina component installs a default codec if both, <code>codec</code> is null and <code>textline</code> 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.
<code>disconnectOnNoReply</code>	<code>true</code>	Camel 2.3: If sync is enabled then this option dictates MinaConsumer if it should disconnect where there is no reply to send back.
<code>noReplyLogLevel</code>	<code>WARN</code>	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: <code>FATAL</code> , <code>ERROR</code> , <code>INFO</code> , <code>DEBUG</code> , <code>OFF</code> .

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:

As the sample is part of a unit test, we test it by sending some data to it on port 6200.

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.

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.

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:

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:

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:

And then we can refer to our endpoint directly in the route, as follows:

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:

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`.

For instance, the example below will send a keep-alive message after 10 seconds of inactivity:

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.



If using the `SslFilter` you need to add the `mina-filter-ssl` JAR to the classpath.

Then, you can configure your endpoint using Spring DSL:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [MINA2](#)
- [Netty](#)

MOCK 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 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

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

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

Options

Option	Default	Description
<code>reportGroup</code>	<code>null</code>	A size to use a throughput logger for reporting

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.

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:

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:

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:

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:

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. Suppose you have the given route below:

Listing 1. Route

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:

Listing 1. adviceWith mocking all endpoints

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:



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.



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:

Listing 1. adviceWith mocking only log endpoints using a pattern

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.

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.

Listing 1. isMockEndpoints using camel-test kit

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:

Listing 1. camel-route.xml



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.

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:

Listing 1. test-camel-route.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:

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`".

Listing 1. adviceWith mock and skip sending to endpoints

The same example using the Test Kit

Listing 1. isMockEndpointsAndSkip using camel-test kit

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.

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`.

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:

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

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

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

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [Spring Testing](#)
- [Testing](#)



time units

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

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:

Note that the Jing component also supports RelaxNG Compact Syntax

URI format

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

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).

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:

```
<code></code>
```

URI format

```
<code></code>
```

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

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	An integer to define the maximum messages to gather per poll. By default, no maximum is set. Can be used to set a limit of e.g. 1000 to avoid when starting up the server that there are thousands of files. Set a value of 0 or negative to 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. simple . simple executor does nothing special. reuse executor reuses prepared statements. batch executor reuses statements and batches updates.

Message Headers

Camel will populate the result message, either IN or OUT with a header with the statement used:

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:

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:

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:

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`:

And the same for **UPDATE**, where we can send an `Account` object as the IN body to MyBatis:

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:

Then you can insert multiple rows, by sending a Camel message to the `mybatis` endpoint which uses the `InsertList` statement type, as shown below:

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:

Then you can update multiple rows, by sending a Camel message to the `mybatis` endpoint which uses the `UpdateList` statement type, as shown below:

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:

Then you can delete multiple rows, by sending a Camel message to the `mybatis` endpoint which uses the `DeleteList` statement type, as shown below:

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

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:

And the MyBatis SQL mapping file used:

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.

```
route().consumeAccount("UPDATE account SET status='processed' WHERE id=${body.id}");
```

And the statements in the sqlmap file:

```
<update id="consumeAccount">
    UPDATE account SET status='processed' WHERE id=${body.id}
</update>
```

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).

```
DataSource dataSource = ...;
DataSourceProxy proxy = new DataSourceProxy(dataSource);
```

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`:

```
TransactionManager transactionManager = new DataSourceTransactionManager(proxy);
```

A `mybatis-spring SqlSessionFactoryBean` then wraps that same `DataSource`:

```
SqlSessionFactoryBean sessionFactoryBean = new SqlSessionFactoryBean();
```

The camel-mybatis component is then configured with that factory:

```
MybatisComponent mybatisComponent = new MybatisComponent();
```

Finally, a transaction policy is defined over the top of the transaction manager, which can then be used as usual:

```
TransactionPolicy transactionPolicy = new TransactionPolicy(transactionManager);
```

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

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:

```
<code></code>
```

URI format

```
<code></code>
```

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	Ê	The port number of the host.
password	Ê	Password to be authenticated when sending checks to Nagios.
connectionTimeout	5000	Connection timeout in millis.
timeout	5000	Sending timeout in millis.
nagiosSettings	Ê	To use an already configured <code>com.googlecode.jsendnsca.core.NagiosSettings</code> object. Then any of the other options are not in use, if using this.
sendSync	true	Whether or not to use synchronous when sending a passive check. Setting it to <code>false</code> 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.
CamelNagiosLevel	This is the severity level. You can use values <code>CRITICAL</code> , <code>WARNING</code> , <code>OK</code> . Camel will by default use <code>OK</code> .
CamelNagiosServiceName	The servie name. Will default use the <code>CamelContext</code> 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:

```
<code></code>
```

To send a `CRITICAL` message you can send the headers such as:

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:

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

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:

URI format

The URI scheme for a netty component is as follows

This component supports producer and consumer endpoints for both TCP and UDP.

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

Options

Name	Default Value	Description
keepAlive	true	Setting to ensure socket is not closed due to inactivity
tcpNoDelay	true	Setting to improve TCP protocol performance
backlog	0	Camel 2.9.6/2.10.4/2.11.1: Allows to configure a backlog for netty consumer (server). Note the backlog is just a best effort depending on the OS. Setting this option to a value such as 200, 500 or 1000, tells the TCP stack how long the "accept" queue can be. If this option is not configured, then the backlog depends on OS setting.
broadcast	false	Setting to choose Multicast over UDP
connectTimeout	10000	Time to wait for a socket connection to be available. Value is in mills.
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. false then the Asynchronous Routing Engine is used, true to force processing synchronous.
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 consumer will enrich the Camel Message with headers having information about the client certificate such as 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. Size is bytes.
receiveBufferSize	65536 bytes	The TCP/UDP buffer sizes to be used during inbound communication. Size is bytes.
option.XXX	null	Camel 2.11/2.10.4: Allows to configure additional netty options using "option." as prefix. For example "option.child.keepAlive=false" to set the netty option "child.keepAlive=false". See the Netty documentation for possible options that can be used.
corePoolSize	10	The number of allocated threads at component startup. Defaults to 10. Note: This option is removed from Camel 2.9.2 onwards. As we rely on Netty's default settings.
maxPoolSize	100	The maximum number of threads that may be allocated to this endpoint. Defaults to 100. Note: This option is removed from Camel 2.9.2 onwards. As we rely on Netty's default settings.
disconnect	false	Whether or not to disconnect(close) from Netty Channel right after use. Can be used for both consumer and producer.
lazyChannelCreation	true	Channels can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started.
transferExchange	false	Only used for TCP. You can transfer the exchange over the wire instead of just the body. The following fields are transferred: In body, Out body, fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level.
disconnectOnNoReply	true	If sync is enabled then this option dictates NettyConsumer if it should disconnect where there is no reply to send back.
noReplyLogLevel	WARN	If sync is enabled this option dictates NettyConsumer which logging level to use when logging a there is no reply to send back. Values are: FATAL, ERROR, INFO, DEBUG, OFF.
serverExceptionCaughtLogLevel	WARN	Camel 2.11.1: If the server (NettyConsumer) catches an exception then its logged using this logging level.
serverClosedChannelExceptionCaughtLogLevel	DEBUG	Camel 2.11.1: If the server (NettyConsumer) catches an <code>java.nio.channels.ClosedChannelException</code> then its logged using this logging level. This is used to avoid logging the closed channel exceptions, as clients can disconnect abruptly and then cause a flood of closed exceptions in the Netty server.
allowDefaultCodec	true	Camel 2.4: The netty component installs a default codec if both, encoder/deocder is null and textline is false. Setting allowDefaultCodec to false prevents the netty component from installing a default codec as the first element in the filter chain.
textline	false	Camel 2.4: Only used for TCP. If no codec is specified, you can use this flag to indicate a text line based codec; if not specified or the value is false, then Object Serialization is assumed over TCP.
delimiter	LINE	Camel 2.4: The delimiter to use for the textline codec. Possible values are LINE and NULL.
decoderMaxLineLength	1024	Camel 2.4: The max line length to use for the textline codec.

autoAppendDelimiter	true	Camel 2.4: Whether or not to auto append missing end delimiter when sending using the textline codec.
encoding	null	Camel 2.4: The encoding (a charset name) to use for the textline codec. If not provided, Camel will use the JVM default Charset.
workerCount	null	Camel 2.9: When netty works on nio mode, it uses default workerCount parameter from Netty, which is <code>cpu_core_threads*2</code> . User can use this operation to override the default workerCount from Netty
sslContextParameters	null	Camel 2.9: SSL configuration using an <code>org.apache.camel.util.jsse.SSLContextParameters</code> instance. See Using the JSSE Configuration Utility.
receiveBufferSizePredictor	null	Camel 2.9: Configures the buffer size predictor. See details at Jetty documentation and this mail thread.
requestTimeout	0	Camel 2.11.1: Allows to use a timeout for the Netty producer when calling a remote server. By default no timeout is in use. The value is in milli seconds, so eg 30000 is 30 seconds.
needClientAuth	false	Camel 2.11: Configures whether the server needs client authentication when using SSL.
orderedThreadPoolExecutor	true	Camel 2.10.2: Whether to use ordered thread pool, to ensure events are processed orderly on the same channel. See details at the netty javadoc of <code>org.jboss.netty.handler.execution.OrderedMemoryAwareThreadPoolExecutor</code> for more details.
maximumPoolSize	16	Camel 2.10.2: The core pool size for the ordered thread pool, if its in use.
producerPoolEnabled	true	Camel 2.10.4/Camel 2.11: Producer only. Whether producer pool is enabled or not.
producerPoolMaxActive	-1	Camel 2.10.3: Producer only. Sets the cap on the number of objects that can be allocated by the pool (checked out to clients, or idle awaiting checkout) at a given time. Use a negative value for no limit.
producerPoolMinIdle	0	Camel 2.10.3: Producer only. Sets the minimum number of instances allowed in the producer pool before the evictor thread (if active) spawns new objects.
producerPoolMaxIdle	100	Camel 2.10.3: Producer only. Sets the cap on the number of "idle" instances in the pool.
producerPoolMinEvictableIdle	30000	Camel 2.10.3: Producer only. Sets the minimum amount of time (value in millis) an object may sit idle in the pool before it is eligible for eviction by the idle object evictor.
bootstrapConfiguration	null	Camel 2.12: Consumer only. Allows to configure the Netty ServerBootstrap options using a <code>org.apache.camel.component.netty.NettyServerBootstrapConfiguration</code> instance. This can be used to reuse the same configuration for multiple consumers, to align their configuration more easily.
bossPoll	null	Camel 2.12: To use a explicit <code>org.jboss.netty.channel.socket.nio.BossPool</code> as the boss thread pool. For example to share a thread pool with multiple consumers. By default each consumer has their own boss pool with 1 core thread.
workerPool	null	Camel 2.12: To use a explicit <code>org.jboss.netty.channel.socket.nio.WorkerPool</code> as the worker thread pool. For example to share a thread pool with multiple consumers. By default each consumer has their own worker pool with 2 x cpu count core threads.
networkInterface	null	Camel 2.12: Consumer only. When using UDP then this option can be used to specify a network interface by its name, such as <code>eth0</code> to join a multicast group.

Registry based Options

Codec Handlers and SSL Keystores can be enlisted in the Registry, such as in the Spring XML file.

The values that could be passed in, are the following:

Name	Description
passphrase	password setting to use in order to encrypt/decrypt payloads sent using SSH
keyStoreFormat	keystore format to be used for payload encryption. Defaults to "JKS" if not set
securityProvider	Security provider to be used for payload encryption. Defaults to "SunX509" if not set.
keyStoreFile	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 <code>ChannelHandler</code> class that can be used to perform special marshalling of outbound payloads. Must override <code>org.jboss.netty.channel.ChannelDownStreamHandler</code> .
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.

Usage Samples

A UDP Netty endpoint using Request-Reply and serialized object payload

A TCP based Netty consumer endpoint using One-way communication

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

Spring DSL based configuration of endpoint

Using Basic SSL/TLS configuration on the Jetty Component

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:

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.

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.

The lists of codecs need to be added to the Camel's registry so they can be resolved when the endpoint is created.

Spring's native collections support can be used to specify the codec lists in an application context

The bean names can then be used in netty endpoint definitions either as a comma separated list or contained in a List e.g.

or via spring.

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:

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.



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 `CamelNettySSLClientCertSubjectName`.



Read further above about using non shareable encoders/decoders.

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

Listing 1. Using custom pipeline factory

The custom channel pipeline factory can then be added to the registry and instantiated/utilized on a camel route in the following way

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:

Then in the Camel routes we can refer to this worker pools by configuring the `workerPool` option in the [URI] as shown below:



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.

And if we have another route we can refer to the shared worker pool:

... and so forth.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
 - 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:

Installing in plain Apache Karaf

In plain Karaf the nmr component can also be installed using the servicemix artifacts:

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:

NMR consumer and producer endpoints

The following code:

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:

The messages sent by this producer endpoint are sent to the already deployed NMR endpoint.

URI format

URI Options

Option	Default Value	Description
<code>runAsSubject</code>	<code>false</code>	Apache ServiceMix 4.4: When this is set to <code>true</code> on a consumer endpoint, the endpoint will be invoked on behalf of the <code>Subject</code> that is set on the <code>Exchange</code> (i.e. the call to <code>Subject.getSubject(AccessControlContext)</code> will return the <code>Subject</code> instance)
<code>synchronous</code>	<code>false</code>	When this is set to <code>true</code> on a consumer endpoint, an incoming, synchronous NMR <code>Exchange</code> 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>throwExceptionOnFailure</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.

Examples

Consumer

Producer

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.

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.

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).

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

URI format

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&...`



If you are using Quartz 2.x then from Camel 2.12 onwards there is a Quartz2 component you should use

Options

Parameter	Default	Description
cron	None	Specifies a cron expression (not compatible with the trigger.* or job.* options).
trigger.repeatCount	0	SimpleTrigger: How many times should the timer repeat?
trigger.repeatInterval	0	SimpleTrigger: The amount of time in milliseconds between repeated triggers.
job.name	null	Sets the job name.
job.XXX	null	Sets the job option with the XXX setter name.
trigger.XXX	null	Sets the trigger option with the XXX setter name.
stateful	false	Uses a Quartz StatefulJob instead of the default job.
fireNow	false	New to Camel 2.2.0, if it is true will fire the trigger when the route is start when using SimpleTrigger.
deleteJob	true	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 pre-configured trigger with camel Uri. Just ensure the names match. Notice you cannot have both deleteJob and pauseJob set to true.
pauseJob	false	Camel 2.12: If set to true, then the trigger automatically pauses when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse pre-configured trigger with camel Uri. Just ensure the names match. Notice you cannot have both deleteJob and pauseJob set to true.

For example, the following routing rule will fire two timer events to the `mock:results` endpoint:

```

route()
    .timer(1000, "timer1")
    .timer(2000, "timer2")
    .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.

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
properties	null	Properties	Camel 2.4: You can configure a <code>java.util.Properties</code> instance.
propertiesFile	null	String	Camel 2.4: File name of the properties to load from the classpath

To do this you can configure this in Spring XML as follows

```

<quartz:quartz>
    <properties>
        <property name="quartz.scheduler.class" value="org.quartz.simpl.SimpleThreadPoolScheduler" />
    </properties>
</quartz:quartz>

```

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
-----------	---------	------	-------------



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.

```
startDelayedSeconds  0          int          Camel 2.4: Seconds to wait before starting the quartz scheduler.
autoStartScheduler   true       boolean      Camel 2.4: Whether or not the scheduler should be auto started.
```

To do this you can configure this in Spring XML as follows

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 node 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:

which is equivalent to using the cron expression

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:

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

URI format

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:

Example URIs:



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.

ENDPOINTS

FIX sessions are endpoints for the **quickfix** component. An endpoint URI may specify a single session or all sessions managed by a specific QuickFIX/J engine. Typical applications will use only one FIX engine but advanced users may create multiple FIX engines by referencing different configuration files in **quickfix** component endpoint URIs.

When a consumer does not include a session ID in the endpoint URI, it will receive exchanges for all sessions managed by the FIX engine associated with the configuration file specified in the URI. If a producer does not specify a session in the endpoint URI then it must include the session-related fields in the FIX message being sent. If a session is specified in the URI then the component will automatically inject the session-related fields into the FIX message.

Exchange Format

The exchange headers include information to help with exchange filtering, routing and other processing. The following headers are available:

Header Name	Description
EventCategory	One of AppMessageReceived, AppMessageSent, AdminMessageReceived, AdminMessageSent, SessionCreated, SessionLogon, SessionLogoff. See the QuickfixjEventCategory enum.
SessionID	The FIX message SessionID
MessageType	The FIX MsgType tag value
DataDictionary	Specifies a data dictionary to used for parsing an incoming message. Can be an instance of a data dictionary or a resource path for a QuickFIX/J data dictionary file

The DataDictionary header is useful if string messages are being received and need to be parsed in a route. QuickFIX/J requires a data dictionary to parse certain types of messages (with repeating groups, for example). By injecting a DataDictionary header in the route after receiving a message string, the FIX engine can properly parse the data.

QuickFIX/J Configuration Extensions

When using QuickFIX/J directly, one typically writes code to create instances of logging adapters, message stores and communication connectors. The **quickfix** component will automatically create instances of these classes based on information in the configuration file. It also provides defaults for many of the common required settings and adds additional capabilities (like the ability to activate JMX support).

The following sections describe how the **quickfix** component processes the QuickFIX/J configuration. For comprehensive information about QuickFIX/J configuration, see the QFJ user manual.

Communication Connectors

When the component detects an initiator or acceptor session setting in the QuickFIX/J configuration file it will automatically create the corresponding initiator and/or acceptor connector. These settings can be in the default or in a specific session section of the configuration file.

Session Setting	Component Action
ConnectionType=initiator	Create an initiator connector
ConnectionType=acceptor	Create an acceptor connector

The threading model for the QuickFIX/J session connectors can also be specified. These settings affect all sessions in the configuration file and must be placed in the settings default section.

Default/Global Setting	Component Action
ThreadModel=ThreadPerConnector	Use SocketInitiator or SocketAcceptor (default)
ThreadModel=ThreadPerSession	Use ThreadedSocketInitiator or ThreadedSocketAcceptor

Logging

The QuickFIX/J logger implementation can be specified by including the following settings in the default section of the configuration file. The `ScreenLog` is the default if none of the following settings are present in the configuration. It's an error to include settings that imply more than one log implementation. 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.

Default/Global Setting	Component Action
ScreenLogShowEvents	Use a ScreenLog
ScreenLogShowIncoming	Use a ScreenLog
ScreenLogShowOutgoing	Use a ScreenLog
SLF4J*	Camel 2.6+. Use a SLF4JLog. Any of the SLF4j settings will cause this log to be used.
FileLogPath	Use a FileLog
JdbcDriver	Use a JdbcLog

Message Store

The QuickFIX/J message store implementation can be specified by including the following settings in the default section of the configuration file. The `MemoryStore` is the default if none of the following settings are present in the configuration. It's an error to include settings that imply more than one message store implementation. 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.

Default/Global Setting	Component Action
JdbcDriver	Use a JdbcStore
FileStorePath	Use a FileStore
SleepycatDatabaseDir	Use a SleepcatStore

Message Factory

A message factory is used to construct domain objects from raw FIX messages. The default message factory is `DefaultMessageFactory`. However, advanced applications may require a custom message factory. This can be set on the QuickFIX/J component.

JMX

Default/Global Setting	Component Action
UseJmx	if <code>Y</code> , then enable QuickFIX/J JMX

Other Defaults

The component provides some default settings for what are normally required settings in QuickFIX/J configuration files. `SessionStartTime` and `SessionEndTime` default to "00:00:00", meaning the session will not be automatically started and stopped. The `HeartBtInt` (heartbeat interval) defaults to 30 seconds.

Minimal Initiator Configuration Example

```
-----
```

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.

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"	QuickfixjProducer.CORRELATION_CRITERIA_KEY	None
Correlation Timeout in Milliseconds	"CorrelationTimeout"	QuickfixjProducer.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 `CompID` fields will be reversed when looking for the reply.

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.

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.

Exception handling

QuickFIX/J behavior can be modified if certain exceptions are thrown during processing of a message. If a `RejectLogon` exception is thrown while processing an incoming logon administrative message, then the logon will be rejected.

Normally, QuickFIX/J handles the logon process automatically. However, sometimes an outgoing logon message must be modified to include credentials required by a FIX counterparty. If the FIX logon message body is modified when sending a logon message (`EventCategory=AdminMessageSent` the modified message will be sent to the counterparty. It is important that the outgoing logon message is being processed *synchronously*. If it is processed asynchronously (on another thread), the FIX engine will immediately send the unmodified outgoing message when 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.

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.

The trader session consumes execution report messages from the market and processes them.

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

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.

Samples

Direction : to FIX gateway

```
-----
```

Direction : from FIX gateway

```
-----
```

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

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:

```
-----
```


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.

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

Options

Name	Default Value	Description
mediaSize	MediaSizeName.NA_LETTER	Sets the stationary as defined by enumeration settings in the javax.print.attribute.standard.MediaSizeName API. The default setting is to use North American Letter sized stationary
copies	1	Sets number of copies based on the javax.print.attribute.standard.Copies API
sides	Sides.ONE_SIDED	Sets one sided or two sided printing based on the javax.print.attribute.standard.Sides API
flavor	DocFlavor.BYTE_ARRAY	Sets DocFlavor based on the javax.print.DocFlavor API
mimeType	AUTOSENSE	Sets mimeTypees supported by the javax.print.DocFlavor API
mediaTray	AUTOSENSE	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 is not start with //hostname/printer
sendToPrinter	true	Setting this option to false prevents sending of the print data to the printer

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

Example 2: Printing GIF based payloads on a Remote printer using A4 stationary and one-sided mode

Example 3: Printing JPEG based payloads on a Remote printer using Japanese Postcard stationary and one-sided mode

PROPERTIES COMPONENT

Available as of Camel 2.3

URI format

Where **key** is the key for the property to lookup

Options

Name	Type	Default	Description
cache	boolean	true	Whether or not to cache loaded properties.
locations	String	null	A list of locations to load properties. You can use comma to separate multiple locations. This option will override any default locations and only use the locations from this option.
ignoreMissingLocation	boolean	false	Camel 2.10: Whether to silently ignore if a location cannot be located, such as a properties file not found.
propertyPrefix	String	null	Camel 2.9 Optional prefix prepended to property names before resolution.
propertySuffix	String	null	Camel 2.9 Optional suffix appended to property names before resolution.
fallbackToUnaugmentedProperty	boolean	true	Camel 2.9 If true, first attempt resolution of property name augmented with <code>propertyPrefix</code> and <code>propertySuffix</code> before falling back the plain property name specified. If false, only the augmented property name is searched.
prefixToken	String	{	Camel 2.9 The token to indicate the beginning of a property token.
suffixToken	String	}	Camel 2.9 The token to indicate the end of a property token.

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 URIs. This works much like you would do if using Spring's `<property-placeholder>` tag. However Spring have a limitation which prevents 3rd party frameworks to leverage Spring property placeholders to the fullest. See more at [How do I use Spring Property Placeholder with Camel XML](#).

The property placeholder is generally in use when doing:

- 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



Resolving property from Java code

You can use the method `resolvePropertyPlaceholders` on the `CamelContext` to resolve a property from any Java code.



Bridging Spring and Camel property placeholders

From Camel 2.10 onwards, you can bridge the Spring property placeholder with Camel, see further below for more details.

Syntax

The syntax to use Camel's property placeholder is to use `{{key}}` for example `{{file.uri}}` where `file.uri` is the property key.

You can use property placeholders in parts of the endpoint URI's which for example you can use placeholders for parameters in the URIs.

PropertyResolver

Camel provides a pluggable mechanism which allows 3rd part to provide their own resolver to lookup 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. You can prefix the locations with either:

- `ref:` **Camel 2.4:** to lookup in the Registry
- `file:` to load the from file system
- `classpath:` to load from classpath (this is also the default if no prefix is provided)
- `blueprint:` **Camel 2.7:** to use a specific OSGi blueprint placeholder service

Defining location

The `PropertiesResolver` need to know a location(s) where to resolve the properties.

You can define 1 to many locations. If you define the location in a single String property you can separate multiple locations with comma such as:

```
location=${my.location}
```

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.

For example:

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:`

Where `APP_HOME` is an OS environment.

You can have multiple placeholders in the same location, such as:

Configuring in Java DSL

You have to create and register the `PropertiesComponent` under the name `properties` such as:

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.

Using the `<propertyPlaceholder>` tag makes the configuration a bit more fresh such as:

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:

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.



Specifying the cache option inside XML

Camel 2.10 onwards supports specifying a value for the cache option both inside the Spring as well as the Blueprint XML.

You can also use placeholders as a part of the endpoint uri:

```
-----
```

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:

```
-----
```

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:

```
-----
```

Examples

You can also use property placeholders directly in the endpoint uris without having to use `properties:`.

```
-----
```

And you can use them in multiple wherever you want them:

```
-----
```

You can also your property placeholders when using `ProducerTemplate` for example:

```
-----
```

Example with Simple language

The Simple language now also support using property placeholders, for example in the route below:

```
-----
```

You can also specify the location in the Simple language for example:

```
-----
```

Additional property placeholder supported 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.

The example below has property placeholder in the `<jmxAgent>` tag:

```
-----
```

You can also define property placeholders in the various attributes on the `<camelContext>` tag such as `trace` as shown here:

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. An example of this is given below

Using property placeholders for any kind of attribute in the XML DSL

Available as of Camel 2.7

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 onwards 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".

In our properties file we have the value defined as

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:

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:

Listing 1. Using OSGi blueprint property placeholders in Camel routes



If you use OSGi Blueprint then this only works from **2.11.1** or **2.10.5** onwards.

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.

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:

Listing 1. Explicit referring to a OSGi blueprint placeholder in Camel

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:

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:

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:

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 .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 ServieMix / Karaf then this container



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.

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`.

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:

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 2nd 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:

Using .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:

And in the unit test class we do as follows:

And the `etc/stuff.cfg` configuration file contains

Bridging Spring and Camel property placeholders

Available as of Camel 2.10

The Spring Framework does not allow 3rd 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:

Listing 1. Bridging Spring and Camel property placeholders

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:

Listing 1. Using bridge property placeholders

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. For example:

clashes with Spring property placeholders, and you should use `$simple{ }` to indicate using the Simple language in Camel.

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 onwards, as the Camel test kits, eg `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.

Listing 1. Providing properties from within unit test source

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:

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:

This will lookup the placeholder with they key "name".

You can also add a default value if the key does not exists, such as:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [Jasypt](#) for using encrypted values (eg passwords) in the properties

REF COMPONENT

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

URI format

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:

And you could have a list of endpoints defined in the Registry such as:

Sample

In the sample below we use the `ref` in the URI to reference the endpoint with the spring ID, `endpoint2`:

You could, of course, have used the `ref` attribute instead:

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:

URI format

Format of `restletUrl`:

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&...

Options

Name	Default Value	Description
headerFilterStrategy=#refName	An instance of RestletHeaderFilterStrategy	Use the # notation (headerFilterStrategy=#refName) to reference a header filter strategy in the Camel Registry. The strategy will be plugged into the restlet binding if it is HeaderFilterStrategyAware.
restletBinding=#refName	An instance of DefaultRestletBinding	The bean ID of a RestletBinding object in the Camel Registry.
restletMethod	GET	On a producer endpoint, specifies the request method to use. On a consumer endpoint, specifies that the endpoint consumes only restletMethod requests. The string value is converted to org.restlet.data.Method by the Method.valueOf(String) method.
restletMethods	None	Consumer only Specify one or more methods separated by commas (e.g, restletMethods=post,put) to be serviced by a restlet consumer endpoint. If both restletMethod and restletMethods options are specified, the restletMethod setting is ignored.
restletRealm=#refName	null	The bean ID of the Realm Map in the Camel Registry.
restletUriPatterns=#refName	None	Consumer only Specify one or more URI templates to be serviced by a restlet consumer endpoint, using the # notation to reference a List<String> in the Camel Registry. If a URI pattern has been defined in the endpoint URI, both the URI pattern defined in the endpoint and the restletUriPatterns option will be honored.
throwExceptionOnFailure (2.6 or later)	true	*Producer only * Throws exception on a producer failure.

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.
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.

Message Headers

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 <code>content-type</code> of the response message. If this header is not set, the content type is based on the object type of the OUT message body. In Camel 2.3 onward, if the Content-Type header is specified in the Camel IN message, the value of the header determine the content type for the Restlet request message. 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 <code>DefaultRestletBinding</code> 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 <code>org.restlet.Request</code> object which holds all request details.
CamelRestletResponse	Response	Camel 2.8: The <code>org.restlet.Response</code> object. You can use this to create responses using the API from Restlet. See examples below.
org.restlet.*	Ê	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 <code>CacheDirective</code> 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.

```


```

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`.

```


```

The following sample starts a `direct` endpoint that sends requests to the server on `http://localhost:8080` (that is, our restlet consumer endpoint).

That is all we need. We are ready to send a request and try out the restlet component:

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)

The sample client gets a response like the following:

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:

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.

The `restletUriPatterns=#uriTemplates` option references the `List<String>` bean defined in the Spring XML configuration.

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:

Listing 1. Generating response using Restlet Response API

Configuring max threads on component

To configure the max threads options you must do this on the component, such as:

**Note**

`org.apache.camel.restlet.auth.login` and `org.apache.camel.restlet.auth.password` will not be propagated as Restlet header.

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 URIs) 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`;

```
<include resource="/org/apache/camel/restlet/servlet/beans.xml"/>
```

And add this to your `web.xml`;

```
<servlet>  
    <servlet-name>restlet</servlet-name>  
    <servlet-class>org.apache.camel.restlet.servlet.SpringServerServlet</servlet-class>  
    <init-param>  
        <param-name>camelContext</param-name>  
        <param-value>/camel-context.xml</param-value>  
    </init-param>  
</servlet>
```

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;

```
<pre>[Content of the REST API endpoint]
```

You will need to add dependency on the Spring extension to restlet which you can do in your Maven `pom.xml` file:

```
<code><dependency>  
    <groupId>org.apache.camel</groupId>  
    <artifactId>camel-spring</artifactId>  
</dependency>
```

And you would need to add dependency on the restlet maven repository as well:

```
<code><repository>  
    <id>restlet</id>  
    <url>http://maven.restlet.org</url>  
</repository>
```

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

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:

URI format

For example:

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	Its now possible to use this option from Camel 2.7 ; in the XML DSL. It can be a list of interface names separated by comma.

Using

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

To bind an existing camel processor or service in an RMI registry, define an RMI endpoint as follows:

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:

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:

```
<code></code>
```

Note: The component currently only supports polling (consuming) feeds.

URI format

```
<code></code>
```

Where `rssUri` is the URI to the RSS feed to poll.

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

Options

Property	Default	Description
<code>splitEntries</code>	<code>true</code>	If <code>true</code> , 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 <code>false</code> , Camel obtains a fresh feed on every poll and returns all of the feed's entries.
<code>filter</code>	<code>true</code>	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.
<code>throttleEntries</code>	<code>true</code>	Camel 2.5: Sets whether all entries identified in a single feed poll should be delivered immediately. If <code>true</code> , only one entry is processed per <code>consumer.delay</code> . Only applicable when <code>splitEntries</code> is set to <code>true</code> .
<code>lastUpdate</code>	<code>null</code>	Use in combination with the <code>filter</code> option to block entries earlier than a specific date/time (uses the <code>entry.updated.timestamp</code>). The format is <code>yyyy-MM-ddTHH:MM:ss</code> . Example: <code>2007-12-24T17:45:59</code> .
<code>feedHeader</code>	<code>true</code>	Specifies whether to add the ROME <code>SyndFeed</code> object as a header.
<code>sortEntries</code>	<code>false</code>	If <code>splitEntries</code> is <code>true</code> , this specifies whether to sort the entries by updated date.
<code>consumer.delay</code>	<code>60000</code>	Delay in milliseconds between each poll.
<code>consumer.initialDelay</code>	<code>1000</code>	Milliseconds before polling starts.
<code>consumer.userFixedDelay</code>	<code>false</code>	Set to <code>true</code> to use fixed delay between polls, otherwise fixed rate is used. See <code>ScheduledExecutorService</code> in JDK for details.

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`.

Option	Value	Behavior
--------	-------	----------



Camel-rss internally uses a patched version of ROME hosted on ServiceMix to solve some OSGi class loading issues.

```
splitEntries true ..... A single entry from the current feed is set in the exchange.
splitEntries false ..... The entire list of entries from the current feed is set in the exchange.
```

Message Headers

Header	Description
CamelRssFeed	The entire SyndFeed 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:

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:

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:

The custom bean for this would be:

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
 - Atom

Unable to render {include} Couldn't find a page to include called: Scalate



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")
```

SEDA COMPONENT

The **sed** 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.

URI format

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
size	Ê	Ê	The maximum capacity of the SEDA 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 onwards, the size is unbounded by default. Notice: Mind if you use this option, then its the first endpoint being created with the queue name, that determines the size. To make sure all endpoints use same size, then configure the size option on all of them, or the first endpoint being created. From Camel 2.11 onwards, 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.
concurrentConsumers	Ê	1	Number of concurrent threads processing exchanges.
waitForTaskToComplete	Ê	IfReplyExpected	Option to specify whether the caller should wait for the async task to complete or not before continuing. The following three options are supported: Always, Never or IfReplyExpected. The first two values are self-explanatory. The last value, IfReplyExpected, will only wait if the message is Request Reply based. The default option is IfReplyExpected. See more information about Async messaging.
timeout	Ê	30000	Timeout (in milliseconds) before a SEDA producer will stop waiting for an asynchronous task to complete. See <code>waitForTaskToComplete</code> and <code>Async</code> for more details. In Camel 2.2 you can now disable timeout by using 0 or a negative value.
multipleConsumers	2.2	false	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 SEDA queue and have each consumer receive a copy of the message. When enabled, this option should be specified on every consumer endpoint.



Synchronous

The Direct component provides synchronous invocation of any consumers when a producer sends a message exchange.

<code>limitConcurrentConsumers</code>	2.3	<code>true</code>	Whether to limit the number of <code>concurrentConsumers</code> to the maximum of 500. By default, an exception will be thrown if a SEDA endpoint is configured with a greater number. You can disable that check by turning this option off.
<code>blockWhenFull</code>	2.9	<code>false</code>	Whether a thread that sends messages to a full SEDA 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.
<code>queueSize</code>	2.9	<code>0</code>	Component only: The maximum default size (capacity of the number of messages it can hold) of the SEDA queue. This option is used if <code>size</code> is not in use.
<code>pollTimeout</code>	2.9.3	<code>1000</code>	Consumer only: The timeout used when polling. 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.
<code>purgeWhenStopping</code>	2.11.1	<code>false</code>	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.
<code>queue</code>	2.12.0	<code>null</code>	Define the queue instance which will be used by seda endpoint
<code>queueFactory</code>	2.12.0	<code>null</code>	Define the QueueFactory which could create the queue for the seda endpoint
<code>failIfNoConsumers</code>	2.12.0	<code>false</code>	Whether the producer should fail by throwing an exception, when sending to a SEDA queue with no active consumers.

Choosing BlockingQueue implementation

Available as of Camel 2.12

By default, the SEDA component always instantiates `LinkedBlockingQueue`, but you can use different implementation, you can reference your own `BlockingQueue` implementation, in this case the `size` option is not used

Or you can reference a `BlockingQueueFactory` implementation, 3 implementations are provided `LinkedBlockingQueueFactory`, `ArrayBlockingQueueFactory` and `PriorityBlockingQueueFactory`:

Use of Request Reply

The SEDA component supports using Request Reply, where the caller will wait for the Async route to complete. For instance:

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** onwards, which allows you to chain as many endpoints as you like.

Concurrent consumers

By default, the SEDA endpoint uses a single consumer thread, but you can configure it to use concurrent consumer threads. So instead of thread pools you can use:

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:

Can wind up with two `BlockQueues`: one from the SEDA endpoint, and one from the workqueue of the thread pool, which may not be what you want. Instead, you might wish to configure a Direct endpoint with a thread pool, which can process messages both synchronously and asynchronously. For example:

You can also directly configure number of threads that process messages on a SEDA endpoint using the `concurrentConsumers` option.

Sample

In the route below we use the SEDA queue to send the request to this async queue to be able to send a fire-and-forget message for further processing in another thread, and return a constant reply in this thread to the original caller.

Here we send a Hello World message and expects the reply to be OK.

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.

Since we have specified **multipleConsumers=true** on the seda foo endpoint we can have those two consumers receive their own copy of the message as a kind of pub-sub style messaging.

As the beans are part of an unit test they simply send the message to a mock endpoint, but notice how we can use `@Consume` to consume from the seda queue.

Extracting queue information.

If needed, information such as queue size, etc. can be obtained without using JMX in this fashion:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [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:

URI format

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



Stream

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.

Options

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>matchOnUriPrefix</code>	<code>false</code>	Whether or not the <code>CamelServlet</code> should try to find a target consumer by matching the URI prefix, if no exact match is found.
<code>servletName</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.

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

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:

```
<servlet-name>myServletName</servlet-name>
```

And in your Camel endpoints then include the servlet name as well

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:

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

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:

Then you can define your route as follows:

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:

Sample when using OSGi

From **Camel 2.6.0**, you can publish the `CamelHttpTransportServlet` as an OSGi service with help of SpringDM like this.

Then use this service in your camel route like this:

For versions prior to Camel 2.6 you can use an `Activator` to publish the `CamelHttpTransportServlet` on the OSGi platform



From Camel 2.7 onwards it's easier to use Servlet in Spring web applications. See Servlet Tomcat Example for details.



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").
```

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [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:

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

Instantiating a ShiroSecurityPolicy Object

A `ShiroSecurityPolicy` object is instantiated as follows

ShiroSecurityPolicy Options

Name	Default Value	Type	Description
iniResourcePath or ini	none	Resource String or Ini Object	A mandatory Resource String for the iniResourcePath or an instance of an Ini object must be passed to the security policy. Resources can be acquired from the file system, classpath, or URLs when prefixed with "file:", "classpath:", or "url:" respectively. For e.g "classpath:shiro.ini"
passPhrase	An AES 128 based key	byte[]	A passPhrase to decrypt ShiroSecurityToken(s) sent along with Message Exchanges
alwaysReauthenticate	true	boolean	Setting to ensure re-authentication on every individual request. If set to false, the user is authenticated and locked such 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 Permissions list is provided to the ShiroSecurityPolicy object, then authorization is deemed as not required
cipherService	AES	org.apache.shiro.crypto.CipherService	Shiro ships with AES & Blowfish based CipherServices. You may use one these or pass in your own Cipher implementation
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 <code>ShiroSecurityTokenInjector</code> as well.

Applying Shiro Authentication on a Camel Route

The `ShiroSecurityPolicy`, tests and permits incoming message exchanges containing an 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.

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.

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

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

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:

You can also provide the username and password in two different headers as shown below:

When you use the username and password headers, then the `ShiroSecurityPolicy` in the Camel route will automatic transform those into a single header with key `ShiroSecurityConstants.SHIRO_SECURITY_TOKEN` with the token. Then token is either a `ShiroSecurityToken` instance, of 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:

```


```

URI format

The URI scheme for a sip endpoint is as follows:

```


```

This component supports producer and consumer endpoints for both TCP and UDP.

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

Options

The SIP Component offers an extensive set of configuration options & capability to create custom stateful headers needed to propagate state via the SIP protocol.

Name	Default Value	Description
stackName	NAME_NOT_SET	Name of the SIP Stack instance associated with an SIP Endpoint.
transport	tcp	Setting for choice of transport potocol. Valid choices are "tcp" or "udp".
fromUser	É	Username of the message originator. Mandatory setting unless a registry based custom FromHeader is specified.
fromHost	É	Hostname of the message originator. Mandatory setting unless a registry based FromHeader is specified
fromPort	É	Port of the message originator. Mandatory setting unless a registry based FromHeader is specified
toUser	É	Username of the message receiver. Mandatory setting unless a registry based custom ToHeader is specified.
toHost	É	Hostname of the message receiver. Mandatory setting unless a registry based ToHeader is specified
toPort	É	Portname of the message receiver. Mandatory setting unless a registry based ToHeader is specified
maxforwards	0	the number of intermediaries that may forward the message to the message receiver. Optional setting. May alternatively be set using as registry based MaxForwardsHeader
eventId	É	Setting for a String based event Id. Mandatory setting unless a registry based FromHeader is specified
eventHeaderName	É	Setting for a String based event Id. Mandatory setting unless a registry based FromHeader is specified
maxMessageSize	1048576	Setting for maximum allowed Message size in bytes.
cacheConnections	false	Should connections be cached by the SipStack to reduce cost of connection creation. This is useful if the connection is used for long running conversations.
consumer	false	This setting is used to determine whether the kind of header (FromHeader,ToHeader etc) that needs to be created for this endpoint
automaticDialogSupport	off	Setting to specify whether every communication should be associated with a dialog.
contentType	text	Setting for contentType can be set to any valid MimeType.
contentSubType	xml	Setting for contentSubType can be set to any valid MimeSubType.
receiveTimeoutMillis	10000	Setting for specifying amount of time to wait for a Response and/or Acknowledgement can be received from another SIP stack
useRouterForAllUris	false	This setting is used when requests are sent to the Presence Agent via a proxy.
msgExpiration	3600	The amount of time a message received at an endpoint is considered valid
presenceAgent	false	This setting is used to 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). Consumers have to set this flag to true.

Registry based Options

SIP requires a number of headers to be sent/received as part of a request. These SIP header can be enlisted in the Registry, such as in the Spring XML file.

The values that could be passed in, are the following:

Name	Description
fromHeader	a custom Header object containing message originator settings. Must implement the type javax.sip.header.FromHeader
toHeader	a custom Header object containing message receiver settings. Must implement the type javax.sip.header.ToHeader
viaHeaders	List of custom Header objects of the type javax.sip.header.ViaHeader. Each ViaHeader containing a proxy address for request forwarding. (Note this header is automatically updated by each proxy when the request arrives at its listener)
contentTypeHeader	a custom Header object containing message content details. Must implement the type javax.sip.header.ContentTypeHeader
callIdHeader	a custom Header object containing call details. Must implement the type javax.sip.header.CallIdHeader
maxForwardsHeader	a custom Header object containing details on maximum proxy forwards. This header places a limit on the viaHeaders possible. Must implement the type javax.sip.header.MaxForwardsHeader
eventHeader	a custom Header object containing event details. Must implement the type javax.sip.header.EventHeader

<code>contactHeader</code>	an optional custom Header object containing verbose contact details (email, phone number etc). Must implement the type <code>javax.sip.header.ContactHeader</code>
<code>expiresHeader</code>	a custom Header object containing message expiration details. Must implement the type <code>javax.sip.header.ExpiresHeader</code>
<code>extensionHeader</code>	a custom Header object containing user/application specific details. Must implement the type <code>javax.sip.header.ExtensionHeader</code>

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"

Creating a Camel SIP Subscriber

In the example below, a SIP Subscriber is created to receive SIP Event publications sent to a user "johndoe@localhost:5154"

- using a SIP Stack named Subscriber
- registering with a Presence Agent user called agent@localhost:5152
- using a registry based eventHeader called evtHdrName. The evtHdrName contains the Event which is se to "Event_A"
- using a registry based eventId called evtId

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

Available 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 is used.

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:

```
<code></code>
```

URI format

```
<code></code>
```

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 will try to use `SSLSocket` to init a connection to the server.

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

URI Options

Name	Default Value	Description
password	password	Specifies the password to use to log in to the SMSC.
systemType	cp	This parameter is used to categorize the type of ESME (External Short Message Entity) that is binding to the SMSC (max. 13 characters).
dataCoding	0	<p>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:</p> <ul style="list-style-type: none"> 0: SMSC Default Alphabet 3: Latin I (ISO-8859-1) 4: Octet unspecified (8-bit binary) 8: UCS2 (ISO/IEC-10646) 13: Extended Kanji JIS(X 0212-1990)
alphabet	0	<p>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 alphabets are:</p> <ul style="list-style-type: none"> 0: SMSC Default Alphabet 4: 8 bit Alphabet 8: UCS2 Alphabet
encoding	ISO-8859-1	only for SubmitSm, ReplaceSm and SubmitMulti Defines the encoding scheme of the short message user data.
enquireLinkTimer	5000	Defines the interval in milliseconds between the confidence checks. The confidence check is used to test the communication path between an ESME and an SMSC.
transactionTimer	10000	Defines the maximum period of inactivity allowed after a transaction, after which an SMPP entity may assume that the session is no longer active. This timer may be active on either communicating SMPP entity (i.e. SMSC or ESME).
initialReconnectDelay	5000	Defines the initial delay in milliseconds after the consumer/producer tries to reconnect to the SMSC, after the connection was lost.
reconnectDelay	5000	Defines the interval in milliseconds between the reconnect attempts, if the connection to the SMSC was lost and the previous was not succeed.
registeredDelivery	1	<p>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:</p> <ul style="list-style-type: none"> 0: No SMSC delivery receipt requested. 1: SMSC delivery receipt requested where final delivery outcome is success or failure. 2: SMSC delivery receipt requested where the final delivery outcome is delivery failure.

		The service type parameter can be used to indicate the SMS Application service associated with the message. The following generic service_types are defined: CMT: Cellular Messaging CPT: Cellular Paging VMN: Voice Mail Notification VMA: Voice Mail Alerting WAP: Wireless Application Protocol USSD: Unstructured Supplementary Services Data
serviceType	CMT	
sourceAddr	1616	Defines the address of SME (Short Message Entity) which originated this message.
destAddr	1717	only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
sourceAddrTon	0	Defines the type of number (TON) to be used in the SME originator address parameters. The following TON values are defined: 0: Unknown 1: International 2: National 3: Network Specific 4: Subscriber Number 5: Alphanumeric 6: Abbreviated
destAddrTon	0	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 sourceAddrTon values defined above.
sourceAddrNpi	0	Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. The following NPI values are defined: 0: Unknown 1: ISDN (E163/E164) 2: Data (X.121) 3: Telex (F.69) 6: Land Mobile (E.212) 8: National 9: Private 10: ERMES 13: Internet (IP) 18: WAP Client Id (to be defined by WAP Forum)
destAddrNpi	0	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 sourceAddrNpi values defined above.
priorityFlag	1	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 0 (lowest) priority 1: Level 1 priority 2: Level 2 priority 3: Level 3 (highest) priority
replaceIfPresentFlag	0	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 replace if present flag values are defined: 0: Don't replace 1: Replace
typeOfNumber	0	Defines the type of number (TON) to be used in the SME. Use the sourceAddrTon values defined above.
numberingPlanIndicator	0	Defines the numeric plan indicator (NPI) to be used in the SME. Use the sourceAddrNpi values defined above.
lazySessionCreation	false	Camel 2.8 onwards Sessions can be lazily created to avoid exceptions, if the SMSC is not available when the Camel producer is started. Camel 2.11 onwards Camel will check the in message headers 'CamelSmppSystemId' and 'CamelSmppPassword' of the first exchange. If they are present, Camel will use these data to connect to the SMSC.
httpProxyHost	null	Camel 2.9.1: If you need to tunnel SMPP through a HTTP proxy, set this attribute to the hostname or ip address of your HTTP proxy.
httpProxyPort	3128	Camel 2.9.1: If you need to tunnel SMPP through a HTTP proxy, set this attribute to the port of your HTTP proxy.
httpProxyUsername	null	Camel 2.9.1: If your HTTP proxy requires basic authentication, set this attribute to the username required for your HTTP proxy.
httpProxyPassword	null	Camel 2.9.1: If your HTTP proxy requires basic authentication, set this attribute to the password required for your HTTP proxy.
sessionStateListener	null	Camel 2.9.3: You can refer to a <code>org.jsmpp.session.SessionStateListener</code> in the Registry to receive callbacks when the session state changed.
addressRange	" "	Camel 2.11: You can specify the address range for the <code>SmppConsumer</code> as defined in section 5.2.7 of the SMPP 3.4 specification. The <code>SmppConsumer</code> will receive messages only from SMSC's which target an address (MSISDN or IP address) within this range.

You can have as many of these options as you like.

```

}

```


Producer Message Headers

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

Header	Type	Description
CamelSmppDestAddr	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.
CamelSmppDestAddrTon	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 sourceAddrTon URI option values defined above.
CamelSmppDestAddrNpi	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 URI option sourceAddrNpi values defined above.
CamelSmppSourceAddr	String	Defines the address of SME (Short Message Entity) which originated this message.
CamelSmppSourceAddrTon	Byte	Defines the type of number (TON) to be used in the SME originator address parameters. Use the sourceAddrTon URI option values defined above.
CamelSmppSourceAddrNpi	Byte	Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. Use the URI option sourceAddrNpi values defined above.
CamelSmppServiceType	String	The service type parameter can be used to indicate the SMS Application service associated with the message. Use the URI option serviceType settings above.
CamelSmppRegisteredDelivery	Byte	only for SubmitSm, ReplaceSm, SubmitMulti and DataSm Is used to request an SMSC delivery receipt and/or SME originated acknowledgements. Use the URI option registeredDelivery settings above.
CamelSmppPriorityFlag	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.
CamelSmppScheduleDeliveryTime	Date	only for SubmitSm, SubmitMulti and ReplaceSm This parameter specifies the scheduled time at which the message delivery should be first attempted. It defines either the absolute date and time or relative time from the current SMSC time at which delivery of this message will be attempted by the SMSC. It can be specified in either absolute time format or relative time format. The encoding of a time format is specified in chapter 7.1.1.1 in the smpp specification v3.4.
CamelSmppValidityPeriod	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 it's provided as Date, it's interpreted as absolute time. Camel 2.9.1 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.
CamelSmppReplaceIfPresentFlag	Byte	only for SubmitSm and SubmitMulti The replace if present flag parameter is used to request the SMSC to replace a previously submitted message, that is still pending delivery. The SMSC will replace an existing message provided that the source address, destination address and service type match the same fields in the new message. The following values are defined: 0: Don't replace 1: Replace
CamelSmppAlphabet / CamelSmppDataCoding	Byte	Camel 2.5 For SubmitSm, SubmitMulti and ReplaceSm (Prior to Camel 2.9 use CamelSmppDataCoding instead of CamelSmppAlphabet.) The data coding according to the SMPP 3.4 specification, section 5.2.19. Use the URI option alphabet settings above.
CamelSmppOptionalParameters	Map<String, String>	Deprecated and will be removed in Camel 2.13.0/3.0.0 Camel 2.10.5 and 2.11.1 onwards and only for SubmitSm, SubmitMulti and DataSm The optional parameters send back by the SMSC.
CamelSmppOptionalParameter	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.COctetString byte[] -> org.jsmpp.bean.OptionalParameter.OctetString Byte -> org.jsmpp.bean.OptionalParameter.Byte Integer -> org.jsmpp.bean.OptionalParameter.Int Short -> org.jsmpp.bean.OptionalParameter.Short null -> org.jsmpp.bean.OptionalParameter.Null

The following message headers are used by the SMPP producer to set the response from the SMSC in the message header

Header	Type	Description
CamelSmppId	List<String>/String	The id to identify the submitted short message(s) for later use. From Camel 2.9.0: In case of a ReplaceSm, QuerySm, CancelSm and DataSm this header vaule is a String. In case of a SubmitSm or SubmitMultiSm this header vaule is a List<String>.
CamelSmppSentMessageCount	Integer	From Camel 2.9 onwards only for SubmitSm and SubmitMultiSm The total number of messages which has been sent.
CamelSmppError	Map<String, List<Map<String, Object>>>	From Camel 2.9 onwards only for SubmitMultiSm The errors which occurred by sending the short message(s) the form Map<String, List<Map<String, Object>>> (messageID : (destAddr : address, error : errorCode)).

CamelSmppOptionalParameters	Map<String, String>	<p>Deprecated and will be removed in Camel 2.13.0/3.0.0</p> <p>From Camel 2.11.1 onwards only for DataSm The optional parameters which are returned from the SMSC by sending the message.</p>
CamelSmppOptionalParameter	Map<Short, Object>	<p>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 <code>Short</code> code for the optional parameter. The value is converted in the following way:</p> <p>org.jsmpp.bean.OptionalParameter.COctetString -> String org.jsmpp.bean.OptionalParameter.OctetString -> byte[] org.jsmpp.bean.OptionalParameter.Byte -> Byte org.jsmpp.bean.OptionalParameter.Int -> Integer org.jsmpp.bean.OptionalParameter.Short -> Short org.jsmpp.bean.OptionalParameter.Null -> null</p>

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
CamelSmppSequenceNumber	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 preserve this field.
CamelSmppCommandId	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 5.1.2.1 in the smpp specification v3.4.
CamelSmppSourceAddr	String	only for AlertNotification, DeliverSm and DataSm Defines the address of SME (Short Message Entity) which originated this message.
CamelSmppSourceAddrNpi	Byte	only for AlertNotification and DataSm Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. Use the URI option <code>sourceAddrNpi</code> values defined above.
CamelSmppSourceAddrTon	Byte	only for AlertNotification and DataSm Defines the type of number (TON) to be used in the SME originator address parameters. Use the <code>sourceAddrTon</code> URI option values defined above.
CamelSmppEsmeAddr	String	only for AlertNotification Defines the destination ESME address. For mobile terminated messages, this is the directory number of the recipient MS.
CamelSmppEsmeAddrNpi	Byte	only for AlertNotification Defines the numeric plan indicator (NPI) to be used in the ESME originator address parameters. Use the URI option <code>sourceAddrNpi</code> values defined above.
CamelSmppEsmeAddrTon	Byte	only for AlertNotification Defines the type of number (TON) to be used in the ESME originator address parameters. Use the <code>sourceAddrTon</code> URI option values defined above.
CamelSmppId	String	only for smsc DeliveryReceipt and DataSm The message ID allocated to the message by the SMSC when originally submitted.
CamelSmppDelivered	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.
CamelSmppDoneDate	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.
CamelSmppFinalStatus	DeliveryReceiptState	<p>only for smsc DeliveryReceipt: The final status of the message. The following values are defined:</p> <p>DELIVRD: Message is delivered to destination EXPIRED: Message validity period has expired. DELETED: Message has been deleted. UNDELIV: Message is undeliverable ACCEPTD: Message is in accepted state (i.e. has been manually read on behalf of the subscriber by customer service) UNKNOWN: Message is in invalid state REJECTD: Message is in a rejected state</p>
CamelSmppCommandStatus	Integer	only for DataSm The Command status of the message.
CamelSmppError	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.
CamelSmppSubmitDate	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 original message was replaced. The format is as follows: YYMMDDhhmm.

CamelSmppSubmitted	Integer	only for smsc DeliveryReceipt Number of short messages originally submitted. This is only relevant when the original message was submitted to a distribution list. The value is padded with leading zeros if necessary.
CamelSmppDestAddr	String	only for DeliverSm and DataSm: Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
CamelSmppScheduleDeliveryTime	String	only for DeliverSm: This parameter specifies the scheduled time at which the message delivery should be first attempted. It defines either the absolute date and time or relative time from the current SMSC time at which delivery of this message will be attempted by the SMSC. It can be specified in either absolute time format or relative time format. The encoding of a time format is specified in Section 7.1.1.1 in the smpp specification v3.4.
CamelSmppValidityPeriod	String	only for DeliverSm 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 7.1.1.1 in the smpp specification v3.4.
CamelSmppServiceType	String	only for DeliverSm and DataSm The service type parameter indicates the SMS Application service associated with the message.
CamelSmppRegisteredDelivery	Byte	only for DataSm Is used to request a delivery receipt and/or SME originated acknowledgements. Same values as in Producer header list above.
CamelSmppDestAddrNpi	Byte	only for DataSm Defines the numeric plan indicator (NPI) in the destination address parameters. Use the URI option <code>sourceAddrNpi</code> values defined above.
CamelSmppDestAddrTon	Byte	only for DataSm Defines the type of number (TON) in the destination address parameters. Use the <code>sourceAddrTon</code> URI option values defined above.
CamelSmppMessageType	String	Camel 2.6 onwards: Identifies the type of an incoming message: AlertNotification: an SMSC alert notification DataSm: an SMSC data short message DeliveryReceipt: an SMSC delivery receipt DeliverSm: an SMSC deliver short message
CamelSmppOptionalParameters	Map<String, Object>	Deprecated and will be removed in Camel 2.13.0/3.0.0 Camel 2.10.5 onwards and only for DeliverSm The optional parameters send back by the SMSC.
CamelSmppOptionalParameter	Map<Short, Object>	Camel 2.10.7, 2.11.2 onwards and only for DeliverSm The optional parameters send back by the SMSC. The key is the <code>Short</code> code for the optional parameter. The value is converted in the following way: org.jsmpp.bean.OptionalParameter.COctetString -> String org.jsmpp.bean.OptionalParameter.OctetString -> byte[] org.jsmpp.bean.OptionalParameter.Byte -> Byte org.jsmpp.bean.OptionalParameter.Int -> Integer org.jsmpp.bean.OptionalParameter.Short -> Short org.jsmpp.bean.OptionalParameter.Null -> null

Exception handling

This component supports the general Camel exception handling capabilities.

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 JSMP 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:

```

try {
    // ...
} catch (Exception e) {
    throw new ProcessRequestException(e);
}

```

Please refer to the SMPP specification for the complete list of error codes and their meanings.

Samples

A route which sends an SMS using the Java DSL:

```

// ...

```

A route which sends an SMS using the Spring XML DSL:

```

<route>
    <from uri="sms:1234567890">
        <to uri="sms:9876543210">
            <body>
                Hello World!
            </body>
        </to>
    </from>
</route>

```



JSMPP library

See the documentation of the JSMPP Library for more details about the underlying library.

A route which receives an SMS using the Java DSL:

A route which receives an SMS using the Spring XML DSL:

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:

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:

URI format

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&...`



SMSC simulator

If you need an SMSC simulator for your test, you can use the simulator provided by Logica.

Options

Name	Default Value	Description
type	none	The type of action you want to perform. Actually you can enter here <code>POLL</code> or <code>TRAP</code> . The value <code>POLL</code> will instruct the endpoint to poll a given host for the supplied OID keys. If you put in <code>TRAP</code> you will setup a listener for <code>SNMP Trap Events</code> .
protocol	udp	Here you can select which protocol to use. You can use either <code>udp</code> or <code>tcp</code> .
retries	2	Defines how often a retry is made before canceling the request.
timeout	1500	Sets the timeout value for the request in millis.
snmpVersion	0 (which means SNMPv1)	Sets the snmp version for the request.
snmpCommunity	public	Sets the community octet string for the snmp request.
delay	60 seconds	Defines the delay in seconds between to poll cycles.
oids	none	Defines which values you are interested in. Please have a look at the Wikipedia to get a better understanding. You may provide a single OID or a coma separated list of OIDs. Example: oids="1.3.6.1.2.1.1.3.0.1.3.6.1.2.1.25.3.2.1.5.1.1.3.6.1.2.1.25.3.5.1.1.1.1.3.6.1.2.1.43.5.1.1.1.1"

The result of a poll

Given the situation, that I poll for the following OIDs:

Listing 1. OIDs

The result will be the following:

Listing 1. Result of toString conversion

As you maybe recognized there is one more result than requested....1.3.6.1.2.1.1.0.
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:

Setting up a trap receiver (**Note that no OID info is needed here!**):

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)

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:

```
<code></code>
```

URI format

```
<code></code>
```

Where **defaultChannelName** represents the default channel name which is used by the Spring Integration Spring context. It will equal to the `inputChannel` name for the Spring Integration consumer and the `outputChannel` name for the Spring Integration provider.

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

Options

Name	Type	Description
<code>inputChannel</code>	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.
<code>outputChannel</code>	String	The Spring integration output channel name that is used to send messages to the Spring integration context.
<code>inOut</code>	String	The exchange pattern that the Spring integration endpoint should use. If <code>inOut=true</code> 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:

```
springIntegration: uri="springIntegration:channelName"
```

Or directly using a Spring integration channel name:

```
springIntegration: channelName
```

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:

```
<?xml version="1.0" ?>
<springIntegration xmlns="http://camel.apache.org/spring-integration" />
```

You can bind your source or target to a Camel endpoint as follows:

```
<springIntegration:source uri="springIntegration:channelName" />
<springIntegration:target uri="springIntegration:channelName" />
```

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

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>2.11.0</version>
</dependency>
```

URI format

```
springLdapTemplate: uri="springLdapTemplate:channelName"
```

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
<code>operation</code>	<code>String</code>	The LDAP operation to be performed. Must be one of <code>search</code> , <code>bind</code> , or <code>unbind</code> .
<code>scope</code>	<code>String</code>	The scope of the search operation. Must be one of <code>object</code> , <code>onelevel</code> , or <code>subtree</code> , 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/](http://static.springsource.org/spring-ldap/site/apidocs/org/springframework/ldap/core/LdapTemplate.html#search%28java.lang.String,%20java.lang.String,%20int,%20org.springframework.ldap.core.At)

[LdapTemplate.html#search%28java.lang.String,%20java.lang.String,%20int,%20org.springframework.ldap.core.At](http://static.springsource.org/spring-ldap/site/apidocs/org/springframework/ldap/core/LdapTemplate.html#search%28java.lang.String,%20java.lang.String,%20int,%20org.springframework.ldap.core.At)

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.springldap.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:

```
<code></code>
```

URI format

The URI scheme for this component is as follows

```
<code></code>
```

To expose a web service **mapping-type** needs to be set to any of the following:

Mapping type	Description
rootqname	Offers the option to map web service requests based on the qualified name of the root element contained in the message.
soapaction	Used to map web service requests based on the SOAP action specified in the header of the message.
uri	In order to map web service requests that target a specific URI.
xpathresult	Used to map web service requests based on the evaluation of an XPath expression against the incoming message. The result of the evaluation should match the XPath result specified in the endpoint URI.
beanname	Allows you to reference an <code>org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher</code> object in order to integrate with existing (legacy) endpoint mappings like <code>PayloadRootQNameEndpointMapping</code> , <code>SoapActionEndpointMapping</code> , etc

As a consumer the **address** should contain a value relevant to the specified mapping-type (e.g. a SOAP action, XPath expression). As a producer the address should be set to the URI of the web service your calling upon.

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

Options

Name	Required?	Description
soapAction	No	SOAP action to include inside a SOAP request when accessing remote web services
wsAddressingAction	No	WS-Addressing 1.0 action header to include when accessing web services. The <code>To</code> header is set to the <i>address</i> of the web service as specified in the endpoint URI (default Spring-WS behavior).
expression	Only when <i>mapping-type</i> is <code>xpathresult</code>	XPath expression to use in the process of mapping web service requests, should match the result specified by <code>xpathresult</code>
timeout	No	Camel 2.10: Sets the socket read timeout (in milliseconds) while invoking a webservice using the producer, see <code>URLConnection.setReadTimeout()</code> and <code>CommonsHttpClientMessageSender.setReadTimeout()</code> . Camel 2.12: The built-in message sender <code>HttpComponentsMessageSender</code> is considered instead of <code>CommonsHttpClientMessageSender</code> which has been deprecated, see <code>HttpComponentsMessageSender.setReadTimeout()</code> .



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)

Camel 2.10: Reference to `org.apache.camel.util.jsse.SSLContextParameters` in the Registry. See Using the JSSE Configuration Utility. This option works when using the built-in message sender implementations: `CommonsHttpMessageSender` and `HttpURLConnectionMessageSender`. 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.

Camel 2.12: The built-in message sender `HttpComponentsMessageSender` is considered **instead of** `CommonsHttpMessageSender` which has been deprecated.

sslContextParameters No

Registry based options

The following options can be specified in the registry (most likely a Spring ApplicationContext) and referenced from the endpoint URI using the # notation.

Name	Required?	Description
webServiceTemplate	No	Option to provide a custom WebServiceTemplate. This allows for full control over client-side web services handling; like adding a custom interceptor or specifying a fault resolver, message sender or message factory.
messageSender	No	Option to provide a custom WebServiceMessageSender. For example to perform authentication or use alternative transports
messageFactory	No	Option to provide a custom WebServiceMessageFactory. For example when you want Apache Axiom to handle web service messages instead of SAAJ
transformerFactory	No	Option to override default TransformerFactory. The provided transformer factory must be of type <code>javax.xml.transform.TransformerFactory</code>
endpointMapping	Only when mapping-type is rootqname, soapaction, uri or xpathresult	Reference to an instance of <code>org.apache.camel.component.spring.ws.bean.CamelEndpointMapping</code> in the Registry/ApplicationContext. Only one bean is required in the registry to serve all Camel/Spring-WS endpoints. This bean is auto-discovered by the MessageDispatcher and used to map requests to Camel endpoints based on characteristics specified on the endpoint (like root QName, SOAP action, etc)
messageFilter	No	Option to provide a custom MessageFilter since 2.10.3. For example when you want to process your headers or attachments by your own.

Message headers

Name	Type	Description
CamelSpringWebserviceEndpointUri	String	URI of the web service your 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
CamelSpringWebserviceAddressingAction	URI	Use this header to specify the WS-Addressing action of the message, overrides wsAddressingAction option if present
CamelSpringWebserviceSoapHeader	Source	Camel 2.11.1: Use this header to specify/access the SOAP headers of the message.

ACCESSING WEB SERVICES

To call a web service at `http://foo.com/bar` simply define a route:

And sent a message:

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:

Optionally you can override the endpoint options with header values:

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

We can set the body and header on the Camel Message as follows:

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 `WebSdrviceMessage` response.

Now you can use

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:

1. Simply define the `messageFactory` as is bellow and spring-ws will use MTOM strategy to populate your SOAP message with optimized attachments.

2. Add into your `pom.xml` the following dependencies

3. Add your attachment into the pipeline, for example using a `Processor` implementation.

4. Define endpoint (producer) as usual, for example like this:

5. Now, your producer will generate MTOM message with optimized 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 or a local message filter as follows:

a) the global custom filter that provides the global configuration for all spring-ws endpoints

or

b) the local `messageFilter` directly on the endpoint as follows:

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`:

Using a custom MessageSender and MessageFactory

A custom message sender or factory in the registry can be referenced like this:

Spring configuration:

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

spring-ws-servlet.xml

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`.

The following route will receive web service requests containing the `http://example.com/GetFoo SOAP` action.

The following route will receive all requests sent to `http://example.com/foobar`.

The route below will receive requests that contain the element `<foobar>abc</foobar>` anywhere inside the message (and the default namespace).

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 `PayloadRootQNameEndpointMapping`.



The use of the `beannname` 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 `beannname` 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 `beannname`:

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:

Similarly when *providing* web services, you can unmarshal XML requests to POJO's and marshal the response message back to XML:

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:

URI format

In addition, the `file` and `url` endpoint URIs are supported:

If the `stream:header` URI is specified, the `stream` header is used to find the stream to write to. This option is available only for stream producers (that is, it cannot appear in `from()`).

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

Options

Name	Default Value	Description
<code>delay</code>	0	Initial delay in milliseconds before consuming or producing the stream.
<code>encoding</code>	JVM Default	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>	null	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>	null	When using the <code>stream:file</code> URI format, this option specifies the filename to stream to/from.
<code>url</code>	null	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>	false	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>retry</code>	false	Camel 2.7: will retry opening the file if it's 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>	false	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.

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:

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`.

The following sample demonstrates how to continuously read a file stream (analogous to the UNIX `tail` command):

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.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

STRING TEMPLATE

The **string-template** component allows you to process a message using a String Template. This can be ideal when using Templating to generate responses for requests.

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

URI format

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template.

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

Options

Option	Default	Description
<code>contentCache</code>	<code>false</code>	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.

delimiterStart	null	Since Camel 2.11.1, configuring the variable start delimiter
delimiterEnd	null	Since 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 loads the resource only once and hot-reloading is not possible. This scenario can be used in production when the resource never changes.

StringTemplate Attributes

Camel will provide exchange information as attributes (just a `java.util.Map`) to the string template. The Exchange is transfered as:

key	value
exchange	The Exchange itself.
headers	The headers of the In message.
camelContext	The Camel Context.
request	The In message.
in	The In message.
body	The In message body.
out	The Out message (only for InOut message exchange pattern).
response	The Out message (only for InOut message exchange pattern).

Samples

For example you could use a string template as follows in order to formulate a response to a message:

```

-----

```

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 `$`.

```

-----

```

And the java code is as follows:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)

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:

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

The SQL component uses the following endpoint URI notation:

From Camel 2.11 onwards you can use named parameters by using `# : name` style as shown:

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.

Notice that the standard `?` symbol that denotes the parameters to an SQL query is substituted with the `#` symbol, because the `?` symbol is used to specify options for the endpoint. The `?` symbol replacement can be configured on endpoint basis.

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



In Camel 2.10 or older the SQL component can only be used as producer.
From Camel 2.11 onwards this component can also be a consumer, eg `from()`.



This component can be used as a Transactional Client.

Options

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 <code>DataSource</code> to look up in the registry. Use <code>dataSource=#theName</code> instead.
dataSource	String	null	Camel 2.11: Reference to a <code>DataSource</code> 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 <code>String.replaceAll()</code> operation and no SQL parsing is involved (quoted strings will also change). This replacement is only happening if the endpoint is created using the <code>SqlComponent</code> . If you manually create the endpoint, then use the expected ? sign instead.
template.<xxx>	£	null	Sets additional options on the Spring <code>JdbcTemplate</code> that is used behind the scenes to execute the queries. For instance, <code>template.maxRows=10</code> . For detailed documentation, see the <code>JdbcTemplate</code> 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 <code>org.apache.camel.component.sql.SqlProcessingStrategy</code> to execute queries when the consumer has processed the rows/batch.
prepareStatementStrategy	£	£	Camel 2.11: Allows to plugin to use a custom <code>org.apache.camel.component.sql.SqlPrepareStatementStrategy</code> 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 <code>ScheduledExecutorService</code> 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.
consumer.useIterator	boolean	true	Camel 2.11: SQL consumer only: If true each row returned when polling will be processed individually. If false the entire <code>java.util.List</code> of data is set as the IN body.
consumer.routeEmptyResultSet	boolean	false	Camel 2.11: SQL consumer only: Whether to route a single empty Exchange if there was no data to poll.
consumer.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.
consumer.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.
consumer.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.
consumer.expectedUpdateCount	int	-1	Camel 2.11: SQL consumer only: If using <code>consumer.onConsume</code> 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.
consumer.breakBatchOnConsumeFail	boolean	false	Camel 2.11: SQL consumer only: If using <code>consumer.onConsume</code> and it fails, then this option controls whether to break out of the batch or continue processing the next row from the batch.

<code>alwaysPopulateStatement</code>	<code>boolean</code>	<code>false</code>	Camel 2.11: SQL producer only: If enabled then the <code>populateStatement</code> method from <code>org.apache.camel.component.sql.SqlPrepareStatementStrategy</code> is always invoked, also if there is no expected parameters to be prepared. When this is <code>false</code> then the <code>populateStatement</code> 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.
<code>separator</code>	<code>char</code>	<code>,</code>	Camel 2.11.1: The separator to use when parameter values is taken from message body (if the body is a <code>String</code> type), to be inserted at <code>#</code> placeholders. Notice if you use named parameters, then a <code>Map</code> type is used instead.
<code>outputType</code>	<code>String</code>	<code>SelectList</code>	Camel 2.12.0: Make the output of consumer or producer to <code>SelectList</code> as List of Map, or <code>SelectOne</code> as single Java object in the following way: a) If the query has only single column, then that JDBC Column object is 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 that result. c) If the <code>outputClass</code> 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 a non-unique result exception.
<code>outputClass</code>	<code>String</code>	<code>null</code>	Camel 2.12.0: Specify the full package and class name to use as conversion when <code>outputType=SelectOne</code> .
<code>parametersCount</code>	<code>int</code>	<code>0</code>	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.
<code>noop</code>	<code>boolean</code>	<code>false</code>	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

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.

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`.

Header values

When performing `update` operations, the SQL Component stores the update count in the following message headers:

Header	Description
--------	-------------

CamelSqlUpdateCount	The number of rows updated for <code>update</code> operations, returned as an <code>Integer</code> object.
CamelSqlRowCount	The number of rows returned for <code>select</code> operations, returned as an <code>Integer</code> object.
CamelSqlQuery	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 <code>?</code> instead of a <code>#</code> symbol

Configuration

You can now set a reference to a `DataSource` in the URI directly:

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:

The SQL script `createAndPopulateDatabase.sql` we execute looks like as described below:

Then we configure our route and our `sql` component. Notice that we use a `direct` endpoint in front of the `sql` endpoint. This allows us to send an exchange to the `direct` endpoint with the URI, `direct:simple`, which is much easier for the client to use than the long `sql:URI`. Note that the `DataSource` is looked up in the registry, so we can use standard Spring XML to configure our `DataSource`.

And then we fire the message into the `direct` endpoint that will route it to our `sql` component that queries the database.

We could configure the `DataSource` in Spring XML as follows:

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:

Though if the message body is a `java.util.Map` then the named parameters will be taken from the body.

Using the JDBC based idempotent repository

Available as of Camel 2.7: In this section we will use the JDBC based 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:

In **Camel 2.8**, we added the `createdAt` column:

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:

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

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
<code>createTableIfNotExists</code>	<code>true</code>	Defines whether or not Camel should try to create the table if it doesn't exist.
<code>tableExistsString</code>	<code>SELECT 1 FROM CAMEL_MESSAGEPROCESSED WHERE 1 = 0</code>	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.



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.

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 (String) and the second one is the message id (String).
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 (String), the second one is the message id (String) 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 (String) and the second one is the message id (String).

A customized

`org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository`
could look like:

Using the JDBC based aggregation repository

Available as of Camel 2.6

`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
<code>dataSource</code>	<code>DataSource</code>	Mandatory: The <code>java.sql.DataSource</code> to use.
<code>repositoryName</code>	<code>String</code>	Mandatory: The name of the repository.
<code>transactionManager</code>	<code>TransactionManager</code>	Mandatory: The <code>org.springframework.transaction.TransactionManager</code> to manage transactions for the repository.
<code>lobHandler</code>	<code>LobHandler</code>	A <code>org.springframework.jdbc.support.LobHandler</code> to use. If this option is not set, Camel will use a <code>DefaultLobHandler</code> .
<code>returnOldExchange</code>	<code>boolean</code>	Whether the get operation should return the old exchange or not. To optimize as we do not want to store the old exchange.
<code>useRecovery</code>	<code>boolean</code>	Whether or not recovery is enabled. If recovery is enabled, automatic recovery will be used if the repository fails.
<code>recoveryInterval</code>	<code>long</code>	If recovery is enabled, the interval in milliseconds between recovery attempts. By default this is 10000.
<code>maximumRedeliveries</code>	<code>int</code>	Allows you to limit the number of redeliveries. If the number of redeliveries exceeds this value, the Exchange will be moved to the dead letter queue. If this option is used then the <code>deadLetterUri</code> must be set.
<code>deadLetterUri</code>	<code>String</code>	An endpoint uri for a <code>DeadLetterChannel</code> to use. If this option is used then the <code>maximumRedeliveries</code> must be set.
<code>storeBodyAsText</code>	<code>boolean</code>	Camel 2.11: Whether to store the body as text. If <code>false</code> , the body will be stored as binary.
<code>headersToStoreAsText</code>	<code>List<String></code>	Camel 2.11: Allows to specify the headers to store as text. If the headers are not in the list, they will be stored in binary format.
<code>optimisticLocking</code>	<code>false</code>	Camel 2.12: To turn on optimistic locking. If <code>true</code> , the repository will use optimistic locking to avoid multiple Camel applications updating the same message.



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.

Camel 2.12: Allows to

`org.apache.camel`
to map vendor specific e
`optimisticLocking`

`jdbcOptimisticLockingExceptionMapper` Ê

What is preserved when persisting

`JdbcAggregationRepository` will only preserve any `Serializable` compatible data types. If a data type is not such a type its dropped and a `WARN` is logged. And it only persists the `Message` body and the `Message` headers. The `Exchange` properties are **not** persisted.

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 is set when an `Exchange` is being recovered/redelivered:

Header	Type	Description
<code>Exchange.REDELIVERED</code>	Boolean	Is set to true to indicate the <code>Exchange</code> is being redelivered.
<code>Exchange.REDELIVERY_COUNTER</code>	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 retried until it success.

You can use option `maximumRedeliveries` to limit the maximum number of redelivery attempts for a given recovered Exchange. You must also set the `deadLetterUri` option so Camel knows where to send the Exchange when the `maximumRedeliveries` was hit.

You can see some examples in the unit tests of camel-sql, for example this test.

Database

To be operational, each aggregator uses two table: the aggregation and completed one. By convention the completed has the same name as the aggregation one suffixed with `"_COMPLETED"`. The name must be configured in the Spring bean with the `RepositoryName` property. In the following example aggregation will be used.

The table structure definition of both table are identical: in both case a String value is used as key (**id**) whereas a Blob contains the exchange serialized in byte array. However one difference should be remembered: the **id** field does not have the same content depending on the table.

In the aggregation table **id** holds the correlation Id used by the component to aggregate the messages. In the completed table, **id** holds the id of the exchange stored in corresponding the blob field.

Here is the SQL query used to create the tables, just replace `"aggregation"` with your aggregator repository name.

```
CREATE TABLE aggregation (id VARCHAR(255) PRIMARY KEY, blob BLOB);
```

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:

```
CREATE TABLE aggregation (id VARCHAR(255) PRIMARY KEY, body VARCHAR(255), companyName VARCHAR(255), accountName VARCHAR(255));
```

And then configure the repository to enable this behavior as shown below:

```
jdbcAggregationRepository.setStoreBodyAndHeadersAsText(true);
```

Codec (Serialization)

Since they can contain any type of payload, Exchanges are not serializable by design. It is converted into a byte array to be stored in a database BLOB field. All those conversions are handled by the `JdbcCodec` class. One detail of the code requires your attention: the `ClassLoadingAwareObjectInputStream`.

The `ClassLoadingAwareObjectInputStream` has been reused from the Apache ActiveMQ 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:

```
-----
```

Optimistick 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 optimistick 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 optimistick locking error.

Here is an example, where we define 2 extra FQN class names from the JDBC vendor.

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [JDBC](#)

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:

From Camel 2.9 onwards the Test component is provided directly in the camel-core.

URI format

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
timeout	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:

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

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`.

Options

Name	Default Value	Description
time	null	A <code>java.util.Date</code> the first event should be generated. If using the URI, the pattern expected is: <code>yyyy-MM-dd HH:mm:ss</code> or <code>yyyy-MM-dd'T'HH:mm:ss</code> .
pattern	null	Allows you to specify a custom <code>Date</code> pattern to use for setting the time option using URI syntax.
period	1000	If greater than 0, generate periodic events every period milliseconds.
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. The default value has been changed to 1000 from Camel 2.11 onwards. In older releases the default value is 0.
fixedRate	false	Events take place at approximately regular intervals, separated by the specified period.
daemon	true	Specifies whether or not the thread associated with the timer endpoint runs as a daemon.



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.

```
repeatCount 0
```

Camel 2.8: Specifies a maximum limit of number of fires. So if you set it to 1, the 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
<code>Exchange.TIMER_NAME</code>	String	The value of the <code>name</code> option.
<code>Exchange.TIMER_TIME</code>	Date	The value of the <code>time</code> option.
<code>Exchange.TIMER_PERIOD</code>	long	The value of the <code>period</code> option.
<code>Exchange.TIMER_FIRED_TIME</code>	Date	The time when the consumer fired.
<code>Exchange.TIMER_COUNTER</code>	Long	Camel 2.8: The current fire counter. Starts from 1.

Message Headers

When the timer is fired, it adds the following information as headers to the IN message

Name	Type	Description
<code>Exchange.TIMER_FIRED_TIME</code>	<code>java.util.Date</code>	The time when the consumer fired

Sample

To set up a route that generates an event every 60 seconds:

```
route().timer().timerName("myTimer").timerTime("10:00:00").timerPeriod("60 seconds").timerFiredTime("10:00:00").timerCounter(1).to("myBean")
```

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().timer().timerName("myTimer").timerTime("10:00:00").timerPeriod("60 seconds").timerFiredTime("10:00:00").timerCounter(1).to("myBean")
```

Firing only once

Available as of **Camel 2.8**



Instead of 60000 you can use `period=60s` which is more friendly to read.

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:

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

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:

From Camel 2.9 onwards the Validation component is provided directly in the camel-core.

Options

Option	Default	Description
resourceResolver	null	Camel 2.9: Reference to a <code>org.w3c.dom.ls.LSResourceResolver</code> in the Registry.
useDom	false	Whether <code>DOMSource/DOMResult</code> or <code>SaxSource/SaxResult</code> should be used by the validator.
useSharedSchema	true	Camel 2.3: Whether the Schema instance should be shared or not. This option is introduced to work around a JDK 1.6.x bug. Xerces should not have this issue.
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 against a header.

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).

```
routeFrom("direct:start").toIfValid("mock:valid").toIfInvalid("mock:invalid");
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

VELOCITY

The **velocity** component allows you to process a message using an Apache Velocity template. This can be ideal when using Templating to generate responses for requests.

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>2.10.3</version>  
</dependency>
```

URI format

```
velocity:classpath:template.vm
```

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

Option	Default	Description
--------	---------	-------------

<code>loaderCache</code>	<code>true</code>	Velocity based file loader cache.
<code>contentCache</code>	<code>true</code>	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.
<code>encoding</code>	<code>null</code>	Character encoding of the resource content.
<code>propertiesFile</code>	<code>null</code>	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):

Header	Description
<code>CamelVelocityResourceUri</code>	The <code>templateName</code> as a <code>String</code> object.

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`:

```
.....
```

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 transfered as:

key	value
<code>exchange</code>	The Exchange itself.
<code>exchange.properties</code>	The Exchange properties.
<code>headers</code>	The headers of the In message.
<code>camelContext</code>	The Camel Context instance.
<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 InOut message exchange pattern).
<code>response</code>	The Out message (only for InOut message exchange pattern).

Hot reloading

The Velocity template resource is, by default, hot reloadable for both file and classpath resources (expanded jar). If you set `contentCache=true`, 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.

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

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:

And to use the content cache, e.g. for use in production, where the `.vm` template never changes:

And a file based resource:

In **Camel 2.1** it's possible to specify what template the component should use dynamically via a header, so for example:

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:

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:

And the java code:

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)

- Getting Started

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

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&...`

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`:

And then we receive exchanges in some other Camel context (such as deployed in another `.war` application):

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- SEDA



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:

```
.....
```

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 exmple as follows:

```
.....
```

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:

```
.....
```

URI format

```
.....
```

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
<code>room</code>	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 <code>superman@jabber.org</code> and want to join the <code>krypton</code> room, then the room URL is <code>krypton@conference.jabber.org</code> . Note the conference part. It is not a requirement to provide the full room JID. If the <code>room</code> parameter does not contain the <code>@</code> symbol, the domain part will be discovered and added by Camel
<code>user</code>	User name (without server name). If not specified, anonymous login will be attempted.
<code>password</code>	Password.
<code>resource</code>	XMPP resource. The default is <code>Camel</code> .
<code>createAccount</code>	If <code>true</code> , an attempt to create an account will be made. Default is <code>false</code> .
<code>participant</code>	JID (jabber ID) of person to receive messages. <code>room</code> parameter has precedence over <code>participant</code> .
<code>nickname</code>	Use nickname when joining room. If room is specified and nickname is not, <code>user</code> will be used for the nickname.
<code>serviceName</code>	The name of the service you are connecting to. For Google Talk, this would be <code>gmail.com</code> .

<code>testConnectionOnStartup</code>	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 <code>true</code> .
<code>connectionPollDelay</code>	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 <code>10 seconds</code> .

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`:

```

-----

```

User `superman` to send messages to `joker`:

```

-----

```

Routing example in Java:

```

-----

```

Consumer configuration, which writes all messages from `joker` into the queue, `evil.talk`.

```

-----

```

Consumer configuration, which listens to room messages:

```

-----

```

Room in short notation (no domain part):

```

-----

```

When connecting to the Google Chat service, you'll need to specify the `serviceName` as well as your credentials:

```

-----

```

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:

URI format

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:

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:

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

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
<code>file:com/acme/mytransform.xml</code>	refers to the file <code>com/acme/mytransform.xml</code> on the classpath
<code>file:/foo/bar.xml</code>	refers to the file <code>/foo/bar.xml</code>

`http://.../...` 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:

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.
uriResolver	null	Camel 2.3: Allows you to use a custom <code>javax.xml.transform.URIResolver</code> . Camel will by default use its own implementation <code>org.apache.camel.builder.xml.XsltUriResolver</code> which is capable of loading from classpath.
resultHandlerFactory	null	Camel 2.3: Allows you to use a custom <code>org.apache.camel.builder.xml.ResultHandlerFactory</code> which is capable of using custom <code>org.apache.camel.builder.xml.ResultHandler</code> types.
failOnNullBody	true	Camel 2.3: Whether or not to throw an exception if the input body is null.
deleteOutputFile	false	Camel 2.6: If you have <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.
output	string	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, where as <code>file</code> is streamed directly to a <code>java.io.File</code> . For file you must specify the filename in the IN header with the key <code>Exchange.XSLT_FILE_NAME</code> which is also <code>CamelXsltFileName</code> . Also any paths leading to the filename must be created beforehand, otherwise an exception is thrown at runtime.
contentCache	true	Camel 2.6: Cache for the resource content (the stylesheet file) when it is loaded. If set to <code>false</code> Camel will reload the stylesheet file on each message processing. This is good for development. Note: from Camel 2.9 a cached stylesheet can be forced to reload at runtime via JMX using the <code>clearCachedStylesheet</code> operation.
allowStAX	false	Camel 2.8.3/2.9: Whether to allow using StAX as the <code>javax.xml.transform.Source</code> .
transformerCacheSize	0	Camel 2.9.3/2.10.1: The number of <code>javax.xml.transform.Transformer</code> object that are cached for reuse to avoid calls to <code>Template.newTransformer()</code> .
saxon	false	Camel 2.11: Whether to use Saxon as the <code>transformerFactoryClass</code> . If enabled then the class <code>net.sf.saxon.TransformerFactoryImpl</code> . You would need to add Saxon to the classpath.

Using XSLT endpoints

For example you could use something like

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:

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*.

And the XSLT just needs to declare it at the top level for it to be available:

Spring XML versions

To use the above examples in Spring XML you would use something like

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:

Will lookup the `staff_tempkalte.xml` 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:

Will now be located relative from the starting endpoint, which for example could be:

Which means Camel will locate the file in the **classpath** as `org/apache/camel/component/xslt/staff_template.xml`.

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

Which then will resolve the `xsl` file under `org/apache/camel/component`.

Using xsl:include and default prefix

When using xsl:include such as:

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,

.. then you would have to prefix all your includes with "file:" as well.

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:

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 explicit 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.

Dynamic stylesheets

Available as of Camel 2.9

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.

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 not 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 been 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](#)