MICRO FOCUS®

Artix 5.6.3

Java Router, Deployment Guide

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2015-02-26

Contents

Preface v
- Open Source Project Resources v
- Document Conventions v
- The Artix ESB Documentation Library vi
- Further Information and Product Support vi
  - Information We Need vii
  - Contact information vii

Deploying a Standalone Router 1
- Introduction to Standalone Deployment 1
- Defining a Standalone Main Method 2
- Adding Components to the Camel Context 4
- Adding RouteBuilders to the Camel Context 5
- Running a Standalone Application 7

Deploying into a Spring Container 8
- Introduction to Spring Deployment 8
- Defining a Spring Main Method 9
- Spring Configuration 9
- Running a Spring Application 12

Components 13
- CORBA 13
- CXF Component 13
  - Endpoint URI format 14
  - Address Endpoint URI 14
  - Bean Endpoint URI 17
  - How the data format affects CXF interceptors 20
  - Combining router processors and CXF interceptors 21
  - Creating a message in POJO data format 23
  - Accessing a message in PAYLOAD data format 23
  - Accessing a message in MESSAGE data format 25
  - How to throw a SOAP fault 25
  - How to propagate CXF request and response contexts 26
- File Component 31
- JMS Component 41
- SOAP 55
- Websphere MQ Component 55

Preface

Open Source Project Resources

Apache Incubator CXF

- **Web site:** http://cxf.apache.org/
- **User's list:** `<user@cxf.apache.org>`

Apache Tomcat

- **Web site:** http://tomcat.apache.org/
- **User's list:** `<users@tomcat.apache.org>`

Apache ActiveMQ

- **Web site:** http://activemq.apache.org/
- **User's list:** `<users@activemq.apache.org>`

Apache Camel

- **Web site:** http://camel.apache.org
- **User's list:** `<users@camel.apache.org>`

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`fixed width`	Fixed width (Courier New font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `javax.xml.ws.Endpoint` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `import java.util.logging.Logger;`
`Fixed width italic`	Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/YourUserName`
Italic	*Italic* words in normal text represent emphasis and introduce *new terms*.

Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the **User Preferences** dialog.

Keying conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
`%`	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
`#`	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
`>`	The notation > represents the MS-DOS or Windows command prompt.
`...`	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
`[ ]`	Brackets enclose optional items in format and syntax descriptions.
`{ }`	Braces enclose a list from which you must choose an item in format and syntax descriptions.
`|`	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in {} (braces).

The Artix ESB Documentation Library

For information on the organization of the Artix ESB library, the document conventions used, and where to find additional resources, see ***Using the Artix ESB Library***.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.

- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/artix.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

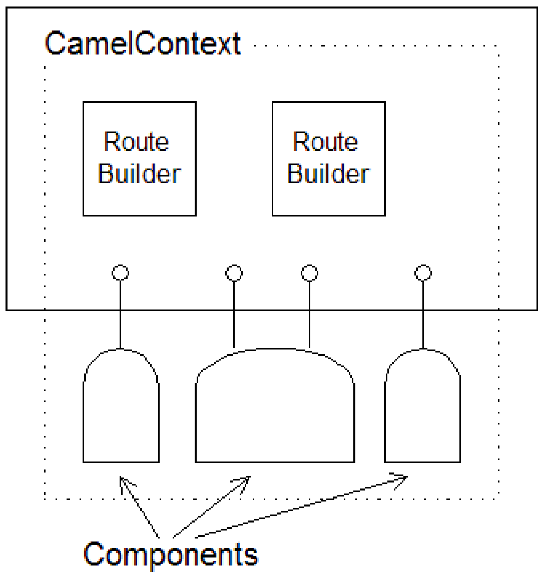
Deploying a Standalone Router

This chapter describes how to deploy the Java Router in standalone mode. This means that you can deploy the router independent of any container, but some extra programming steps are required.

Introduction to Standalone Deployment

Figure 1 gives an overview of the architecture for a router deployed in standalone mode.

Figure 1. Standalone Router



Camel context

The Camel context represents the router service itself. In contrast to most container deployment modes (where the Camel context instance is normally hidden), the standalone deployment requires you to explicitly create and initialize the Camel context in your application code. As part of the initialization procedure, you explicitly create components and route builders and add them to the Camel context.

Components

Components represent connections to particular kinds of destination—for example, a file system, a Web service, a JMS broker, a CORBA service, and so on. In order to read and write messages to and from various destinations, you need to configure and register components, by adding them to the Camel context.

RouteBuilders

The `RouteBuilder` classes represent the core of your router application, because they define the routing rules. In a standalone deployment, you are responsible for managing the lifecycle of `RouteBuilder` objects. In particular, you must create instances of the route builder objects and register them, by adding them to the Camel context.

Defining a Standalone Main Method

In the case of a standalone deployment, it is up to the application developer to create, configure and start a Camel context instance (which encapsulates the core of the router functionality). For this purpose, you should define a `main()` method that performs the following key tasks:

1. Create a Camel context instance.
2. Add components to the Camel context.
3. Add routing rules (RouteBuilder objects) to the Camel context.
4. Start the Camel context, so that it activates the routing rules you defined.

Example of a standalone main method

Example 1 on page 20 shows the standard outline of a standalone `main()` method, which is defined in an example class, `CamelJmsToFileExample`. This example shows how to initialize and activate a Camel context instance.

Example 1. Standalone Main Method

```
package org.apache.camel.example.jmstofile;

import javax.jms.ConnectionFactory;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.camel.CamelContext;
import org.apache.camel.CamelTemplate;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.jms.JmsComponent;
import org.apache.camel.impl.DefaultCamelContext;

public final class CamelJmsToFileExample {

private CamelJmsToFileExample() {
    }

    public static void main(String args[]) throws Exception
{ ❶
        CamelContext context = new DefaultCamelContext(); ❷

        // Add components to the Camel context. ❸
        // ... (not shown)

        // Add routes to the Camel context. ❹
        // ... (not shown)

        // Start the context. context.start(); ❺

        // End of main thread.
    }
}
```

Where the preceding code can be explained as follows:

❶ Define a static `main()` method to serve as the entry point for running the standalone router.

❷ For a standalone router, you need to instantiate a Camel context explicitly. There is just one implementation of CamelContext currently available, the `DefaultCamelContext` class.

❸ The first step in initializing the Camel context is to add any components that your need for your routes (see Adding Components to the Camel Context).

❹ The second step in initializing the Camel context is to add one or more RouteBuilder objects (see Adding RouteBuilders to the Camel Context).

❺ The `CamelContext.start()` method creates a new thread and starts to process incoming messages using the registered routing rules. If the main thread now exits, the Camel context sub-thread remains active and continues to process messages. Typically, you can stop the router by typing `Ctrl-C` in the window where you launched the router application (or by sending a `kill` signal in UNIX). If you want more control over stopping the router process, you could use the `CamelContext.stop()` method in combination with an instrumentation library (such as JMX).

Adding Components to the Camel Context

Relationship between components and endpoints

The essential difference between components and endpoints is that, when configuring a component, you provide concrete connection details (for example, hostname, IP port, and so on), whereas, when specifying an endpoint URI, you provide abstract identifiers (for example, queue name, service name, and so on). It is also possible to define multiple endpoints for each component. For example, a single message broker (represented by a component) can support connections to multiple different queues (represented by endpoints).

The relationship between an endpoint and a component is established through a *URI prefix*. Whenever you add a component to the Camel context, the component gets associated with a particular URI prefix (specified as the first argument to the `CamelContext.addComponent()` method). Endpoint URIs that start with that prefix are then automatically parsed by the associated component.

Example of adding a component

Example 2 shows the outline of the standalone `main()` method, highlighting details of how to add a JMS component to the Camel context.

Example 2. Adding a Component to the Camel Context

```
public final class CamelJmsToFileExample {
    ...
    public static void main(String args[]) throws Exception
{
        CamelContext context = new DefaultCamelContext();

        // Add components to the Camel context.
        ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory("vm://localhost?broker.persistent=false"); ❶

        context.addComponent("test-jms",
JmsComponent.jmsComponentAutoAcknowledge(connectionFactory)); ❷


        // Add routes to the Camel context.
        // ... (not shown)

        // Start the context. context.start();

        // End of main thread.
    }
}
```

Where the preceding code can be explained as follows:

❶ Before you can add a JMS component to the Camel context, you need to create a JMS connection factory (an implementation of `javax.jms.ConnectionFactory`). In this example, the JMS connection factory is implemented by the FUSE Message Broker class, `ActiveMQConnectionFactory`. The broker URL, `vm://localhost`, specifies a broker that is co-located in the same Java Virtual Machine (JVM) as the router. The broker library automatically instantiates the new broker as soon as you try to send a message to it.

❷ Add a JMS component named `test-jms` to the Camel context. This example uses a JMS component with the *auto-acknowledge* option set to true. This implies that messages received from a JMS queue will automatically be acknowledged (receipt confirmed) by the JMS component.

Adding RouteBuilders to the Camel Context

`RouteBuilder` objects represent the core of your router application, because they embody the routing rules you want to implement. In the case of a standalone deployment, you have to manage the lifecycle of your `RouteBuilder` objects explicitly, which involves instantiating the `RouteBuilder` classes and adding them to the Camel context.

Example of adding a RouteBuilder

Example 3 shows the outline of the standalone `main()` method, highlighting details of how to add a `RouteBuilder` object to the Camel context.

Example 3. Adding a RouteBuilder to the Camel Context

```
package org.apache.camel.example.jmstofile;
...
public class JmsToFileRoute extends RouteBuilder { ❶
    public void configure() {
        from("test-jms:queue:test.queue").to("file://test");
 ❷
        // set up a listener on the file component
        from("file://test").process(new Processor() { ❸
            public void process(Exchange e) {
                System.out.println("Received exchange: " + e.getIn());
            }
        });
    }
}

public final class CamelJmsToFileExample {
    ...
    public static void main(String args[]) throws Exception
{
        CamelContext context = new DefaultCamelContext();

        // Add components to the Camel context.
        // ... (not shown)

        // Add routes to the Camel context. context.addRoutes(new
        JmsToFileRoute()); ❹

        // Start the context. context.start();
        // End of main thread.
    }
}
```

Where the preceding code can be explained as follows:

❶ Define a class that inherits from `org.apache.camel.builder.RouteBuilder` in order to define your routing rules. If required, you can define multiple `RouteBuilder` classes.

❷ The first route implements a hop from a JMS queue to the file system. That is, messages are read from the JMS queue `test.queue`, and then written to files in the `test` directory. The JMS endpoint, which has a URI prefixed by `test-jms`, uses the JMS component registered in Example 2.

❸ The second route reads (and deletes) the messages from the `test` directory and displays the messages in the console window. To display the messages, the route implements a custom processor (implemented inline).

❹ Call the `CamelContext.addRoutes()` method to add a `RouteBuilder` object to the Camel context.

Running a Standalone Application

Downloading ActiveMQ

Before running this sample code, you must download ActiveMQ 5.x, and add relevant jar files to the classpath.

Setting the CLASSPATH

Configure your application's CLASSPATH as follows:

- Add *ArtixRoot*/lib/it-soa-router.jar to the CLASSPATH.

Running the application

Assuming that you have coded a `main()` method, as described in Defining a Standalone Main Method, you can run your application using Sun's J2SE interpreter with the following command:

```
java org.apache.camel.example.jmstofile.CamelJmsToFileExample
```

If you are developing the application using a Java IDE (for example, Eclipse (`http://www.eclipse.org/`) or IntelliJ (`http://www.jetbrains.com/idea`)), you can run your application by selecting the `CamelJmsToFileExample` class and directing the IDE to run the class.

Normally, an IDE would automatically choose the static `main()` method as the entry point to run the class.

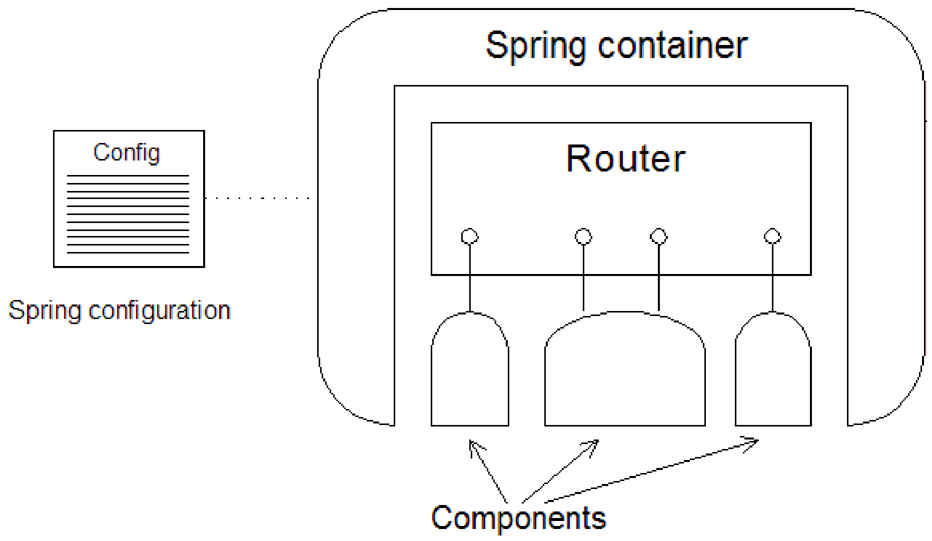
Deploying into a Spring Container

This chapter describes how to deploy the Java Router into a Spring container. A notable feature of the Spring container deployment is that it enables you to specify routing rules in an XML configuration file.

Introduction to Spring Deployment

Figure 2 gives an overview of the architecture for a router deployed into a Spring container.

Figure 2. Router Deployed in a Spring Container



Spring wrapper class

To instantiate a Spring container, Java Router provides the Spring wrapper class, `org.apache.camel.spring.Main`, which exposes methods for creating a Spring container. The wrapper class simplifies the procedure for creating a Spring container, because it includes a lot of boilerplate code required for the router. For example, the wrapper class specifies a default location for the Spring configuration file and adds the Camel context schema to the Spring configuration, enabling you to specify routes using the `camelContext` XML element.

Lifecycle of RouteBuilder objects

The Spring container is responsible for managing the lifecycle of `RouteBuilder` objects. In practice, this means that the router developer need only define the `RouteBuilder` classes. The Spring container will find and instantiate the `RouteBuilder` objects after it starts up (see Spring Configuration).

Spring configuration file

The Spring configuration file is a key feature of the Spring container. Through the Spring configuration file you can instantiate and link together Java objects. You can also configure any Java object using the dependency injection feature.

In addition to these generic features of the Spring configuration file, Java Router defines an extension schema that enables you to define routing rules in XML.

Component configuration

In order to use certain transport protocols in your routes, you must configure the corresponding component and add it to the Camel context. You can add components to the Camel context by defining `bean` elements in the Spring configuration file (see Configuring components).

Defining a Spring Main Method

Java Router defines a convenient wrapper class for the Spring container. To instantiate a Spring container instance, all that you need to do is write a short `main()` method that delegates creation of the container to the wrapper class.

Example of a Spring main method

Example 4 shows how to define a Spring `main()` method for your router application.

Example 4. Spring Main Method

```
package my.package.name;

public class Main {
    public static void main(String[] args) {
        org.apache.camel.spring.Main.main(args);
    }
}
```

Where `org.apache.camel.spring.Main` is the Spring wrapper class, which defines a static `main()` method that instantiates the Spring container.

Spring Configuration

You can use a Spring configuration file to configure the following basic aspects of a router application:

- Specify the Java packages that contain `RouteBuilder` classes.
- Define routing rules in XML.
- Configure components.

In addition to these core aspects of router configuration, you can take advantage of the generic Spring mechanisms for configuring and linking together Java objects within the Spring container.

Location of the Spring configuration file

The Spring configuration file for your router application must be stored at the following location, relative to your CLASSPATH:

```
META-INF/spring/camel-context.xml
```

Basic Spring configuration

Example 5 shows a basic Spring XML configuration file that instantiates and activates `RouteBuilder` classes defined in the *`my.package.name`* Java package.

Example 5. Basic Spring XML Configuration

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Configures the Camel Context-->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=" http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd ❶

       http://camel.apache.org/schema/spring
       http://camel.apache.org/camel/schema/spring/camel-spring.xsd">
 ❷

  <camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
❸
    <package>my.package.name</package> ❹
  </camelContext>
</beans>
```

The preceding configuration can be explained as follows:

❶ This line specifies the location of the Spring framework schema. The URL represents a real, physical location from where you can download the schema. The version of the Spring schema currently supported by Java Router is Spring 3.0.

❷ This line specifies the location of the Camel context schema. The URL shown in this example always points to the latest version of the schema.

❸ Define a `camelContext` element, which belongs to the namespace, `http://camel.apache.org/schema/spring.`

❹ Use the package element to specify one or more Java package names.

As it starts up, the Spring wrapper automatically instantiates and activates any `RouteBuilder` classes that it finds in the specified packages.

Configuring components

To configure router components, use the generic Spring bean configuration mechanism (which implements a *dependency injection* configuration pattern). That is, you define a Spring `bean` element to create a component instance, where the `class` attribute specifies the full class name of the relevant Java Router component. Bean properties on the component class can then be set using the Spring `properties` element. Using the dependency injection mechanism, you can determine what properties you can set by consulting the JavaDoc for the relevant component.

Example 6 shows how to configure a JMS component using Spring configuration. This component configuration enables you to access endpoints of the format `jms:[queue|topic]:`*`QueueOrTopicName`* in your routing rules.

Example 6. Configuring Components in Spring

```
<?xml version="1.0" encoding="UTF-8"?>

<beans ... >

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

    <!-- Java packages (not shown) ... -->
  </camelContext>

  <!-- Configure the default ActiveMQ broker URL -->
  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent"> ❶
    <property name="connectionFactory"> ❷
      <bean class="org.apache.activemq.ActiveMQConnectionFactory"> ❸
         <property name="brokerURL" value="vm://local
host?broker.persistent=false&amp;broker.useJmx=false"/> ❹
       </bean>
    </property>
  </bean>

</beans>
```

Where the preceding configuration can be explained as follows:

❶ Use the `class` attribute to specify the name of the component class—in this example, we are configuring the JMS component class, `JmsComponent`. The `id` attribute specifies the prefix to use for JMS endpoint URIs. For example, with the `id` equal to jms you can connect to an endpoint like `jms:queue:FOO.BAR` in your application code.

❷ When you set the property named, `connectionFactory`, Spring implicitly calls the `JmsComponent.setConnectionFactory()` method to initialize the JMS component at run time.

❸ The connection factory property is initialized to be an instance of `ActiveMQConnectionFactory` (that is, an instance of a FUSE Message Broker message queue).

❹ When you set the `brokerURL` property on `ActiveMQConnectionFactory`, Spring implicitly calls the `setBrokerURL()` method on the connection factory instance. In this example, the broker URL, `vm://localhost`, specifies a broker that is co-located in the same Java Virtual Machine (JVM) as the router. The broker library automatically instantiates the new broker as soon as you try to send a message to it.

For more details about configuring components in Spring, see *Components*.

Running a Spring Application

Downloading ActiveMQ

You must first download ActiveMQ version 5.*x*, and include relevant jar files in the classpath.

Setting the CLASSPATH

Configure your application's CLASSPATH as follows:

1. Add all of the JAR files in *ArtixRoot*`/lib/it-soa-router.jar` to the CLASSPATH.

2. Add the directory containing `META-INF/spring/camel-context.xml` to the CLASSPATH. For example, if your Spring configuration file is `/var/my_router_app/META-INF/spring/camel-context.xml`, add the following directory to the CLASSPATH:

```
/var/my_router_app
```

Running the application

Assuming that you have coded a `main()` method, as described in Defining a Spring Main Method, you can run your application using Sun's J2SE interpreter with the following command:

```
java my.package.name.Main
```

If you are developing the application using a Java IDE (for example, Eclipse (`http://www.eclipse.org/`) or IntelliJ (`http://www.jetbrains.com/idea`)), you can run your application by selecting the *my.package.name*`.Main` class and directing the IDE to run the class.

Normally, an IDE would automatically choose the static `main()` method as the entry point to run the class.

Components

In Java Router, a component is essentially an integration plug-in, which can be used to enable integration with different kinds of protocol, containers, databases, and so on. By adding a component to your Camel context, you gain access to a particular type of endpoint, which can then be used as the sources and targets of your routes. This reference chapter provides an overview of the components available in Java Router.

CORBA

The CORBA protocol does not have a dedicated component. It is supported through the CXF component—see CXF Component.

CXF Component

Introduction to CXF Component

The CXF component enables you to access endpoints using the Apache CXF[1] open services framework (primarily Web services). Because CXF has support for multiple different protocols, you can use a CXF component to access many different kinds of service. For example, CXF supports the following bindings (message encodings):

- SOAP 1.1.
- SOAP 1.2
- CORBA

And CXF supports the following transports:

- HTTP
- RESTful HTTP
- IIOP (transport for CORBA only)
- JMS
- WebSphere MQ

Adding the CXF component

There is no need to add the CXF component to the Camel context; it is automatically loaded by the router core.

Configuring the CXF component to use log4j

The default logger for the CXF component is `java.util.logging`. To configure the CXF component to use the Apache log4j logger instead, perform the following steps:

1. Create a text file named `META-INF/cxf/org.apache.cxf.logger`, with the following contents:

   ```
   org.apache.cxf.common.logging.Log4jLogger
   ```

   This file should contain only this text, on a single line.

2. Add the file to your Classpath, taking care that it precedes the `camel-cxf` JAR file.

Endpoint URI format

There are two different URI formats supported by the CXF component, as follows:

- Address Endpoint URI
- Bean Endpoint URI

Address Endpoint URI

Endpoint URI format

The CXF address endpoint URI conforms to the following format:

```
cxf://Address[?QueryOptions]
```

Where *`Address`* is the physical address of the endpoint, whose format is binding/transport specific (for example, the HTTP URL format, `http://`, for SOAP/HTTP or the corbaloc format, `corbaloc:iiop:`, for CORBA/IIOP). You can optionally add a list of query options, `?`*`QueryOptions`*, in the following format:

```
?Option=Value&Option=Value&Option=Value...
```

URI query options

The CXF URI supports the query options described in Table 1.

Table 1. CXF URI Query Options

Option	Description
`address`	The endpoint address (overriding the value that appears in the first part of the CXF URI).
`dataFormat`	The format used to represent messages internally. Can be one of `POJO`, `PAYLOAD`, or `MESSAGE`.

Option	Description
`serviceClass`	The value of the service class depends on whether the endpoint is a producer or a consumer, as follows: • *Producer endpoint*—the name of the service endpoint interface (SEI). Do not specify a proxy class here. The CXF component will automatically determine the proxy type and create a proxy instance for you. • *Consumer endpoint*—the name of the class that implements the service (which derives from the SEI). If the implementation class is appropriately annotated (following JSE-181 - `http://jcp.org/en/jsr/detail?id=181`), it also determines the WSDL location, service name, and port name for the WSDL endpoint.
`portName`	The port QName (defaults to the value of the annotation in the service class, if one is specified).
`serviceName`	The service QName (defaults to the value of the annotation in the service class, if one is specified). This is required for `camel-cxf` consumer if more than one serviceName is present in the WSDL.
`wsdlURL`	Location of the WSDL contract file (defaults to the value of the annotation in the service class, if one is specified).
`relayHeaders`	*(POJO data format only)* If a route connects a CXF consumer endpoint to a CXF producer endpoint, this boolean option (set on the producer endpoint) determines whether the SOAP headers received from the consumer endpoint are relayed to the producer endpoint and whether SOAP headers set by the producer endpoint are sent back to the consumer endpoint. Default is true (do relay headers). When this option is true, headers can also be filtered by installing custom filters of `MessageHeadersRelay` type. For details, see Filtering Message Headers.
`Wrapped`	Which kind of operation the CXF endpoint producer will invoke. May be `true` or `false`. Default is `false`.
`wrappedStyle`	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. Default is null.
`setDefaultBus`	If `true`, sets the default bus when CXF endpoint creates a bus by itself. Default is `false`.

Option	**Description**
Bus	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`. For example: `bus=#busName`
cxfBinding	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`). For example: `cxfBinding=#bindingName`
headerFilterStrategy	Use # notation to reference a header filter strategy object from the registry. The referenced object must be an instance of `org.apache.camel.spi.HeaderFilterStrategy` (use an instance of `org.apache.camel.component.cxf.CxfHeaderFilterStrategy`) For example: `headerFilterStrategy=#strategyName`
loggingFeatureEnabled	This option enables the CXF Logging Feature which writes inbound and outbound SOAP messages to log. Defaults to `false.` For example: `loggingFeatureEnabled=true`
defaultOperationName	This option sets the default operationName that will be used by the CxfProducer which invokes the remote service. The default is null. For example: `defaultOperationName=greetMe`
defaultOperationNamespace	This option sets the default operationNamespace that will be used by the CxfProducer which invokes the remote service. The default is null. For example: `defaultOperationNamespace=http://apache.org/hello_world_soap_http`
synchronous	This option lets the cxf endpoint decide whether to use the sync or async API to do the underlying work. The default value is `false`, which means that camel-cxf endpoint will try to use the async API by default. For example: synchronous=true
publishedEndpointUrl	This option specifies an endpoint URL that overrides the one published from the WSDL. The endpoint can be accessed with service address URL plus `?wsdl` to see the whole definition of the corresponding wsdl. The default is null. For example: `publishedEndpointUrl=http://example.com/service`

Option	**Description**
`properties.XXX`	Enables you to set custom properties for CXF in the endpoint URL. For example, set `properties.mtom-enabled=true` to enable MTOM.
`allowStreaming`	When the CXF component is running in PAYLOAD mode (see Accessing a message in PAYLOAD data format): • If this option is set to `false`, the CXF component uses the DOM parser to parse the incoming messages into DOM Elements. • If set to `true`, it keeps the payload as a `javax.xml.transform.Source` object that would allow streaming in some cases. This is the default.
`skipFaultLogging`	This option controls whether the PhaseInterceptorChain skips logging the Fault that it catches. Defaults to `false`.
`cxfEndpointConfigurer`	Enables you to programmatically configure the CXF endpoint by applying `org.apache.camel.component.cxf.CxfEndpointConfigurer`. Specify a child class that extends `CxfEndpointConfigurer`.
`username`	Specifies the username used to authenticate the CXF client.
`password`	Specifies the password used to authenticate the CXF client.
`continuationTimeout`	This option is used to set the CXF continuation timeout (in milliseconds) which can be used in CxfConsumer by default when the CXF server is using Jetty or Servlet transport. The default is `30000`. For example: `continuation=80000`

Bean Endpoint URI

Endpoint URI format

The CXF bean endpoint URI conforms to the following format:

```
cxf:bean:BeanID[?QueryOptions]
```

`BeanID` is the ID of a CXF endpoint bean that is registered in the Spring bean registry. To create the associated CXF endpoint bean, add a `cxf:cxfEndpoint` element to your Spring configuration, as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://camel.apache.org//schema/cxf"
       ...>
    ...
    <cxf:cxfEndpoint id="BeanID" serviceClass="serviceClassName"
        address="https://localhost:58001/GreeterService/BasicAuthPort"
        wsdlURL="WsdlLocation"
        endpointName="ns:portName"
        serviceName="ns:serviceName"
        xmlns:ns="XmlNamespace">
     </cxf:cxfEndpoint>
    ...
</beans>
```

You can optionally add a list of query options, *?QueryOptions*—see Table 1 for a list of available options.

cxfEndpoint attributes

The `cxf:cxfEndpoint` element supports the following attributes:

Table 2. Attributes of cxf:cxfEndpoint Element

Attribute	Description
`wsdlURL`	The location of the WSDL contract. Can be a Classpath URL, `classpath:`, file URL, `file:`, or remote URL, `http:`.
`serviceName`	The WSDL service name (from the `name` attribute of the relevant `wsdl:service` element in the WSDL contract). The format of this attribute is *NsPrefix*:*ServiceName*, where *NsPrefix* is a namespace prefix valid at this scope.
`endpointName`	The WSDL endpoint name (from the `name` attribute of the relevant `wsdl:port` element in the WSDL contract). The format of this attribute is *NsPrefix*:*EndpointName*, where *NsPrefix* is a namespace prefix valid at this scope.
`address`	The WSDL endpoint's address, which overrides the value from the WSDL contract.
`bus`	The name of the CXF Bus that provides the context for this JAX-WS endpoint.
`serviceClass`	The class name of the SEI (Service Endpoint Interface) class, which could optionally have JSR181 annotations.

cxfEndpoint child elements

The `cxf:cxfEndpoint` element can optionally contain the following child elements:

Table 3. Child Elements of cxf:cxfEndpoint

Child Element	Description
`cxf:inInterceptors`	The incoming interceptors for this endpoint. A list of `bean` elements or `ref` elements.
`cxf:inFaultInterceptors`	The incoming fault interceptors for this endpoint. A list of `bean` elements or `ref` elements.

Child Element	Description
cxf:outInterceptors	The outgoing interceptors for this endpoint. A list of bean elements or ref elements.
cxf:outFaultInterceptors	The outgoing fault interceptors for this endpoint. A list of bean elements or ref elements.
cxf:properties	A properties map, which sets the JAX-WS endpoint's bean properties. See Using cxf:properties to set endpoint properties.
cxf:handlers	A JAX-WS handler list for the JAX-WS endpoint. See JAX-WS Configuration (http://cxf.apache.org/docs/jax-ws-configuration.html).
cxf:dataBinding	Enables you to specify the DataBinding for this endpoint, where the data binding can be instantiated using the <bean class="MyDataBinding"/> syntax.
cxf:binding	Enables you to specify the BindingFactory for this endpoint, where the binding factory can be instantiated using the <bean class="MyBindingFactory"/> syntax.
cxf:features	The features that hold the interceptors for this endpoint. A list of bean elements or ref elements.
cxf:schemaLocations	The schema locations available to the endpoint. A list of schemaLocation elements.
cxf:serviceFactory	The service factory for this endpoint, where the service factory can be instantiated using the <bean class="MyServiceFactory"/> syntax.

Using cxf:properties to set endpoint properties

You can use the `cxf:properties` child element to set any of the bean properties listed in Table 1. For example, you can set the CXF endpoint's `dataFormat` and `setDefaultBus` bean properties as follows:

```
<cxf:cxfEndpoint id="testEndpoint" address="http://local host:9000/router"
   serviceClass="org.apache.camel.component.cxf.HelloService"

    endpointName="s:PortName" serviceName="s:ServiceName"
    xmlns:s="http://www.example.com/test">
    <cxf:properties>
      <entry key="dataFormat" value="MESSAGE"/>
      <entry key="setDefaultBus" value="true"/>
    </cxf:properties>
</cxf:cxfEndpoint>
```

Programming with CXF Messages

A CXF endpoint allows you to select different data formats for the propagated messages, as shown in Table 4. This subsection describes how to access or modify the different data formats in CXF messages.

Table 4. CXF Data Formats

Data Format	Description
`POJO`	With the *plain old Java object* (POJO) format, the message body contains a list of the Java parameters to the method being invoked on the target server. The type of the POJO message body is `org.apache.cxf.message.MessageContentsList`.
`PAYLOAD`	The message body contains the contents of the `soap:body` element after message configuration in the CXF endpoint is applied. The type of the PAYLOAD message body is `List<org.w3c.dom.Element>`.
`RAW`	The message body contains the raw message that is received from the transport layer. The type of the MESSAGE message body is `InputStream`.
`CXF_MESSAGE`	The CXF_MESSAGE format allows you to invoke the full capabilities of CXF interceptors by converting the message from the transport layer into a raw SOAP message.
`MESSAGE`	This format is deprecated.

How the data format affects CXF interceptors

The choice of data format causes CXF interceptors in certain phases to be skipped. This is unavoidable, for technical reasons. Some CXF interceptor phases are logically incompatible with certain data formats. The choice of data format affects CXF interceptor phases as follows:

- `POJO`—All CXF interceptor phases are processed as normal.
- `PAYLOAD`—CXF interceptor phases are processed, *except* for the following phases:
  - *In phases*—UNMARSHAL, PRE_LOGICAL, PRE_LOGICAL_ENDING, POST_LOGICAL, POST_LOGICAL_ENDING, PRE_INVOKE.
  - *Out phases*—MARSHAL, MARSHAL_ENDING, PRE_LOGICAL, PRE_LOGICAL_ENDING, POST_LOGICAL, POST_LOGICAL_ENDING.
- `RAW`—*Only* the following CXF interceptor phases are processed (all others being skipped):
  - *In phases*—RECEIVE, USER_STREAM, INVOKE, POST_INVOKE.

- *Out phases*—PREPARE_SEND, PREPARE_SEND_ENDING, USER_STREAM, WRITE, SEND.

TIP: For optimum efficiency, select the lowest level data format compatible with the kind of processing you need to perform. The data formats can be ranked in order of efficiency (starting with the most efficient), as follows: RAW, PAYLOAD, POJO.

Combining router processors and CXF interceptors

When designing a route that processes CXF messages, typically the best strategy is to use a combination of router processors *and* CXF interceptors. Each type of processing has its strengths and weaknesses:

- *CXF interceptors* offer the advantage that you can access the message at all levels of marshalling and parsing. For example, you can add a CXF interceptor to process a SOAP message in its raw format and add another interceptor to process the parsed operation parameters.

  By contrast, in a router processor, you can only access the message in the form selected by the data format option.

- *Router processors* enable you to apply the power of the Java DSL to process and route CXF messages. For example, you can easily apply the content based routing pattern to send a CXF message to various endpoints, depending on the contents of a header or an operation parameter.

You also need to remember to take into account the fact that when the PAYLOAD or RAW data formats are selected, some of the CXF interceptor phases are skipped.

Identifying the data format

The easiest way to check the data format in a processor is to look up the `CxfConstants.DATA_FORMAT_PROPERTY` property on the exchange. For example, given an exchange instance, `exchange`, that originates from a CXF endpoint:

```
// Java
import org.apache.camel.component.cxf.common.message.CxfConstants;
...
String dataFormat =
exchange.getProperty(CxfConstants.DATA_FORMAT_PROPERTY).toString();
```

The returned data format can have one of the values: `POJO`, `PAYLOAD`, `CIF_MESSAGE`, or `RAW`.

Accessing a message in POJO data format

The POJO data format is based on the CXF invoker (`http://cxf.apache.org/docs/invokers.html`). The message header has a `CxfConstants.OPERATION_NAME` property, which contains the name of the operation to invoke, and the message body is a list of the SEI method parameters. The following example shows how to access the contents of a POJO message in the implementation of a `Processor`.

```
// Java
public class PersonProcessor implements Processor {

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

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

        BindingOperationInfo boi = (BindingOperationInfo)exchange.getProperty
             (BindingOperationInfo.class.toString());
        if (boi != null) {
        LOG.info("boi.isUnwrapped" + boi.isUnwrapped());
        }
      // Get the parameters list which element is the holder.

      MessageContentsList msgList =
          (MessageContentsList)exchange.getIn().getBody();
      Holder<String> personId = (Holder<String>)msg List.get(0);
      Holder<String> ssn = (Holder<String>)msgList.get(1);
      Holder<String> name = (Holder<String>)msgList.get(2);


      if (personId.value == null || personId.value.length()== 0) {

      LOG.info("person id 123, so throwing exception");

      // Try to throw out the soap fault message
org.apache.camel.wsdl_first.types.UnknownPerson
      Fault personFault = new
org.apache.camel.wsdl_first.types.UnknownPersonFault();
      personFault.setPersonId("");
org.apache.camel.wsdl_first.UnknownPersonFault fault = new
org.apache.camel.wsdl_first.UnknownPersonFault("Get the null value of person
name", personFault);

      // Since camel has its own exception handler framework, we can't throw the
      exception to
      /// trigger it
      // We just set the fault message in the exchange for camel-cxf component
      handling
      // exchange.getFault().setBody(fault);
  }

      name.value = "Bonjour";
      ssn.value = "123";
      LOG.info("setting Bonjour as the response");

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

Creating a message in POJO data format

To create a message in POJO data format, first specify the operation name in the CxfConstants.`OPERATION_NAME` message header. Next, add the method parameters to a list and set the message with this parameter list. The response message's body is of `MessageContentsList` type. For example:

```
// Java
Exchange senderExchange = new DefaultExchange(context, Exchange Pattern.InOut);
final List<String> params = new ArrayList<String>();

// Prepare the request message for the camel-cxf procedure
params.add(TEST_MESSAGE);
senderExchange.getIn().setBody(params);
senderExchange.getIn().setHeader(CxfConstants.OPERATION_NAME,
 ECHO_OPERATION);

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

org.apache.camel.Message out = exchange.getOut();
// The response message's body is an MessageContentsList which first element
// is the return value of the operation,
// If there are some holder parameters, the holder parameter will be filled
// in the reset of List.
// The result will be extract from the MessageContentsList with the String
// class type

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

Accessing a message in PAYLOAD data format

The PAYLOAD format 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.

For example:

```
protected RouteBuilder createRouteBuilder() {
  return new RouteBuilder() {
      public void configure() {
          from(simpleEndpointURI + "&dataFormat=PAYLOAD").to("log:info").process
                (new Processor() {
              @SuppressWarnings("unchecked")
              public void process(final Exchange exchange) throws Exception {
                  CxfPayload<SoapHeader> requestPayload =
                       exchange.getIn().getBody(CxfPayload.class);
                  List<Source> inElements = requestPayload.getBodySources();
                  List<Source> outElements = new ArrayList<Source>();
                  // You can use a customer toStringConverter to turn a CxfPayLoad
                  // message into String if you want
                  String request = exchange.getIn().getBody(String.class);
                  XmlConverter converter = new XmlConverter();
                  String documentString = ECHO_RESPONSE;

                  Element in = new XmlConverter().toDOMElement(inElements.get(0));
                  // Just check the element namespace
                  if (!in.getNamespaceURI().equals(ELEMENT_NAMESPACE)) {
                      throw new IllegalArgumentException("Wrong element namespace");
                  }
                  if (in.getLocalName().equals("echoBoolean")) {
                      documentString = ECHO_BOOLEAN_RESPONSE;
                      checkRequest("ECHO_BOOLEAN_REQUEST", request);
                  } else {
                      documentString = ECHO_RESPONSE;
                      checkRequest("ECHO_REQUEST", request);
                  }
                  Document outDocument = converter.toDOMDocument(documentString);
                  outElements.add(new DOMSource(outDocument.getDocumentElement()));
                  // set the payload header with null
                  CxfPayload<SoapHeader> responsePayload = new
                       CxfPayload<SoapHeader>(null, outElements, null);
                  exchange.getOut().setBody(responsePayload);
              }
          });
      }
  };
}
```

Accessing a message in MESSAGE data format

To access a message in MESSAGE data format, retrieve the message from the underlying CXF message as a `java.io.InputStream` type. For example:

```
// Java
import java.io.InputStream;
...
from(routerEndpointURI).process(new Processor() {
    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception
{
   ...
   InputStream inputStream = exchange.getIn().getBody(InputStream.class);

       // Continue processing the raw message from InputStream

        ...
    }

})
.to(serviceEndpointURI);
```

How to throw a SOAP fault

You can use the `throwFault()` DSL command to throw a SOAP fault, and this works for the `POJO`, `PAYLOAD`, and `RAW` data formats. First of all, you need to define a SOAP fault, as follows:

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

Once you have created the fault, `SOAP_FAULT`, you can throw it as follows:

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

If your CXF endpoint is configured to use the RAW data format, you could set the SOAP Fault message in the message body and set the response code in the message header. For example:

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

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

});
```

How to propagate CXF request and response contexts

The CXF client API provides a way to invoke an operation with request and response context. For example, to set the request context and get the response context for an operation that is invoked through a CXF producer endpoint, you can use code like the following:

```
CxfExchange exchange = (CxfExchange)template.send(getJaxwsEnd pointUri(), new
Processor() {
            public void process(final Exchange exchange) { final List<String> params =
                new ArrayList<String>();
                params.add(TEST_MESSAGE);
                // Set the request context to the inMessage Map<String, Object>
                requestContext = new HashMap<String, Object>();
              requestContext.put(BindingProvider.ENDPOINT_AD DRESS_PROPERTY,
JAXWS_SERVER_ADDRESS);
               exchange.getIn().setBody(params);
               exchange.getIn().setHeader(Client.REQUEST_CONTEXT , requestContext);
               exchange.getIn().setHeader(CxfConstants.OPER ATION_NAME,
               GREET_ME_OPERATION);
              }
        });
        org.apache.camel.Message out = exchange.getOut();
        // The output is an object array, the first element of the array is the return
value
        Object[] output = out.getBody(Object[].class); LOG.info("Received output text:
        " + output[0]);

        // Get the response context form outMessage Map<String, Object>
        responseContext = CastUtils.cast((Map)out.getHeader(Client.RESPONSE_CONTEXT));
        assertNotNull(responseContext);
        assertEquals("Get the wrong wsdl opertion name",
"{http://apache.org/hello_world_soap_http}greetMe",
responseContext.get("javax.xml.ws.wsdl.operation").toString());
```

Filtering Message Headers

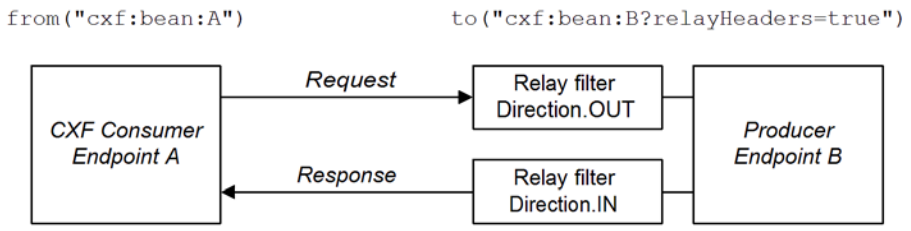
When more than one CXF endpoint appears in a route, you need to decide whether or not to allow headers to propagate between the endpoints. By default, the headers are relayed back and forth between the endpoints, but in many cases it might be necessary to filter the headers or to block them altogether. You can control header propagation by applying filters to producer endpoints (filtering is *not* applicable to consumer endpoints).

The simplest kind of route that can illustrate CXF header filtering is as follows:

```
from("cxf:bean:A").to("cxf:bean:B?relayHeaders=true");
```

In this route, filtering is applied to request headers and response headers before and after entering the producer endpoint, as shown in Figure 3.

Figure 3. Relay Filter Architecture


from("cxf:bean:A")
to("cxf:bean:B?relayHeaders=true")
CXF Consumer Endpoint A
Request
Response
Relay filter Direction.OUT
Relay filter Direction.IN
Producer Endpoint B

In-band headers

An *in-band header* is a header that is explicitly defined as part of the WSDL binding contract for an endpoint.

Out-of-band headers

An *out-of-band header* is a header that is serialized over the wire, but is not explicitly part of the WSDL binding contract. In particular, the SOAP binding permits out-of-band headers, because the SOAP specification does *not* require headers to be defined in the WSDL contract.

Semantics of the relayHeaders option

By default, the `relayHeaders` option is `true` on all CXF producer endpoints. In this case, in-band headers and out-of-band headers are affected differently: in-band headers are all relayed, without exception, while out-of-band headers are subjected to filtering. When the `relayHeaders` option is set explicitly to `false` on a CXF producer endpoint, both in-band headers and out-of-band headers are completely blocked.

The semantics of the `relayHeaders` option can be summarized as follows:

	In-band headers	**Out-of-band headers**
`relayHeaders=true`	*Relay all*	*Filter*
`relayHeaders=false`	*Block*	*Block*

MessageHeaderFilter interface

When the `relayHeaders` option is enabled, out-of-band headers are subject to filtering, where relay filters are implemented by sub-classing the `MessageHeaderFilter` interface, as shown in Example 7.

Example 7. MessageHeaderFilter Interface

```
package org.apache.camel.component.cxf.common.header;
import java.util.List;

import org.apache.camel.spi.HeaderFilterStrategy.Direction;
import org.apache.cxf.headers.Header;

public interface MessageHeaderFilter {
    List<String> getActivationNamespaces();
    void filter(Direction direction, List<Header> headers);
}
```

Implementing the filter() method

The `MessageHeaderFilter.filter()` method is responsible for applying header filtering. Filtering is applied both before and after an operation is invoked on the producer endpoint. Hence, there are two directions to which filtering is applied, as follows:

- `Direction.OUT`

  When the `direction` parameter equals `Direction.OUT`, the filter is processing headers from a Camel message to a CXF message.

- `Direction.IN`

  When the `direction` parameter equals `Direction.IN`, the filter is processing headers from a CXF message to a Camel message.

Binding filters to XML namespaces

It is possible to register multiple relay filters against a given CXF endpoint. The CXF endpoint selects the appropriate filter to use based on the XML namespace of the WSDL binding protocol (for example, the namespace for the SOAP 1.1 binding or for the SOAP 1.2 binding). If a header's namespace is unknown, the `DefaultMessageHeadersRelay` (which relays all headers) is selected by default.

To bind a filter to one or more namespaces, implement the `getActivationNamespaces()` method, which returns the list of bound XML namespaces.

Identifying the namespace to bind to

Example 8 illustrates how to identify the namespaces to which you can bind a filter. This example shows the WSDL file for a Bank server that exposes SOAP endpoints.

Example 8. Sample Binding Namespaces

```
<wsdl:definitions
targetNamespace="http://cxf.apache.org/schemas/cxf/idl/bank"
    xmlns:tns="http://cxf.apache.org/schemas/cxf/idl/bank"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    ...
    <wsdl:binding name="BankSOAPBinding" type="tns:Bank">
       <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="getAccount">
            ...
        </wsdl:operation>
        ...
    </wsdl:binding>
    ...
</wsdl>
```

From the `soap:binding` tag, you can infer that namespace associated with the SOAP binding is `http://schemas.xmlsoap.org/wsdl/soap/`.

Built-in filters

`SoapMessageHeaderFilter` is the built in filter.

- `SoapMessageHeaderFilter`

  This filter is designed to filter standard SOAP headers. It is bound to the following XML namespaces:

```
http://schemas.xmlsoap.org/soap/
http://schemas.xmlsoap.org/wsdl/soap/
http://schemas.xmlsoap.org/wsdl/soap12/
```

Implementing a custom filter

If you want to implement your own custom filter, define a class that inherits from the `MessageHeaderFilter` interface and implement its methods as described in this section. For example, Example 9 shows an example of a custom filter, `CustomHeaderFilter`, that binds to the SOAP namespaces (covering both SOAP 1.1 and SOAP 1.2) and relays all of the headers that pass through it.

Example 9. Sample Relay Filter Implementation

```
//Java
package org.apache.camel.component.cxf.soap.headers;

import java.util.Arrays;
import java.util.List;

import
org.apache.camel.component.cxf.common.header.MessageHeaderFilter;
import org.apache.camel.spi.HeaderFilterStrategy.Direction;
import org.apache.cxf.headers.Header;

public class CustomHeaderFilter implements MessageHeaderFilter {

    public static final String ACTIVATION_NAMESPACE =
"http://cxf.apache.org/bindings/custom";
    public static final List<String> ACTIVATION_NAMESPACES =
Arrays.asList(ACTIVATION_NAMESPACE);


    public List<String> getActivationNamespaces() {
        return ACTIVATION_NAMESPACES;
    }

    public void filter(Direction direction, List<Header> headers) {
    }
}
```

Deploying a custom filter

To apply a custom relay filter to a CXF endpoint, perform the following steps:

1. Create an instance of your custom filter class.
2. Add a `java.util.List` (or any `java.util.Collection` type) containing your custom filter to the `org.apache.camel.cxf.message.headers.relays` endpoint bean property. If you want to apply multiple custom filters, simply add them to the list.

`MessageHeaderFilter` is a property of `CxfHeaderFilterStrategy`. Here is an example of configuring user defined Message Header Filters:

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

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

File Component

The file component provides access to the file system, enabling you to read messages from files and write messages to files. It is useful for simple demonstrations and testing purposes.

Adding the file component

There is no need to add the file component to the Camel context; it is embedded in the router core.

Endpoint URI format

A file endpoint has a URI that conforms to the following format:

`file://`*`FileOrDirectory`*`?`*`QueryOptions`*

`?`*`Option`*`=`*`Value`*`&`*`Option`*`=`*`Value`*`&`*`Option`*`=`*`Value`*`...`

URI query options

The file URI supports the query options described in Table 5.

Table 5. File URI Query Options

Option	Default	Description
Common options		
`autoCreate`	`true`	Automatically create missing directories in the file's pathname. For the file consumer, that means creating the starting directory. For the file producer, it means creating the directory the files should be written to.
`bufferSize`	128kb	The size of the write buffer, in bytes.
`fileName`	`null`	Use an `Expression`, such as `File Expression Language`, to set the filename dynamically. For consumers, this value is used as a filename filter. For producers, it is used to evaluate the filename to write. If an expression is set, it take precedence over the `CamelFileName` header (even if the header itself is an `Expression`). The expression options support both `String` and `Expression` types. If the expression is a `String` type, it is always evaluated using the File Language. If the expression is an `Expression` type, the specified `Expression` type is used - this allows you, for instance, to use OGNL expressions. For the consumer, you can use it to filter filenames, so you can for instance consume today's file using the File Language syntax: `mydata-${date:now:yyyyMMdd}.txt`. The producers support the `CamelOverruleFileName` header which takes precedence over any existing `CamelFileName` header; the `CamelOverruleFileName` is a header that is used only once, and makes it easier as this avoids having to store `CamelFileName` and restore it afterwards.
`flatten`	`false`	Flattens the file name path to strip any leading paths, reducing it to just the file name. This allows you to consume recursively into sub-directories, but when you for example write the files to another directory they will be written in a single directory. Setting this to `true` on the producer enforces that any file name received in the `CamelFileName` header will be stripped of any leading paths.
`charset`	`null`	Specifies the encoding of the file. Use this on the consumer to specify the encodings of the files, so that Camel knows 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.

Option	Default	Description
copyAndDeleteOnRenameFail	true	Whether to fallback and do a copy and delete file, if the file cannot be renamed directly.
renameUsingCopy	false	Perform rename operations using a copy and delete strategy. This is primarily used in environments where the regular rename operation is unreliable (for example, across different file systems or networks). This option takes precedence over the copyAndDeleteOnRenameFail parameter, which falls back to the same copy and delete strategy but only after additional delays.
Consumer only		
initialDelay	1000	Milliseconds before polling of the file/directory starts.
delay	500	Milliseconds before the next poll of the file/directory.
useFixedDelay	false	If true, poll once after the initial delay.
RunLoggingLevel	TRACE	Specifies the logging level for the start/complete log line which the consumer runs when it polls.
recursive	false	If true and the file URI specifies a directory path, the file component polls for changes in all sub-directories.
delete	false	If true, delete the file after processing (the default is to move it).
noop	false	If true, do not move, delete, or modify the file in any way. This option is good for read only data, or for ETL type requirements.
preMove	null	An 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 order directory set this value to order.
move	.camel	An Expression (such as File Language) used to dynamically set the filename when moving it after processing. For example, to move files into a .done subdirectory set this value to .done.
moveFailed	null	An Expression (such as File Language) used to set a different target directory if moving files, as defined by the move option, fails. For example, to move files into a .error subdirectory use: .error. Note that when moving the files to the “fail” location Camel handles the error and will not pick up the file again.
include	null	Used to include files, if the filename matches the regex pattern.
exclude	null	Used to exclude files, if the filename matches the regex pattern.

Option	Default	Description
antInclude	null	Ant-style filter inclusion, for example antInclude=*/.txt. Multiple inclusions may be specified in comma-delimited format.
antExclude	null	Ant-style filter exclusion. If both antInclude and antExclude are used, antExclude takes precedence. Multiple exclusions may be specified in comma-delimited format.
antFilterCaseSensitive	True	Whether Ant-style filters are case-sensitive or not.
idempotent	false	Whether to use the Idempotent Consumer EIP pattern to let Camel skip already-processed files. Will by default use a memory based LRUCache that holds 1000 entries. If noop=true then idempotent is enabled as well to avoid consuming the same files over and over again.
idempotentKey	Expression	To use a custom idempotent key. By default the absolute path of the file is used. You can use File Language. For example to use the file name and file size, specify: idempotentKey=${file:name}-${file:size}
idempotentRepository	null	A pluggable repository org.apache.camel.spi.IdempotentRepository. By default uses MemoryMessageIdRepository if no value is specified and idempotent is true.
inProgressRepository	memory	A pluggable in-progress repository org.apache.camel.spi.IdempotentRepository. The in-progress repository is used to account the current in progress files being consumed. By default a memory-based repository is used.
filter	null	A pluggable filter as a org.apache.camel.component.file.GenericFileFilter class. Will skip files if the filter returns false in its accept() method.
Sorter	null	A pluggable sorter as a java.util.Comparator<org.apache.camel.component.file.GenericFile> class.
sortBy	null	A built-in sort using the File Language. It supports nested sorts, so you can have a sort by file name and as a second group sort by modified date. See http://camel.apache.org/file2.html for details.

Option	Default	Description
readLock	**markerFile**	Used to poll the files if it has exclusive read-lock on the file (that is, if the file is not in-progress or being written). Camel will wait until the file lock is granted. This option provides the following built-in strategies: • markerFile Camel creates a marker file (fileName.camelLock) and then holds a lock on it. This option is **not** available for the FTP component. • changed 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 readLockCheckInterval can be used to set the check frequency. This option is **only** available for the FTP component. Notice that the FTP option fastExistsCheck can be enabled to speedup this readLock strategy, if the FTP server supports the LIST operation with a full file name (some servers may not). • fileLock is for using java.nio.channels.FileLock. This option is **not** available 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. • rename tries to rename the file as a test if it can get an exclusive read-lock. • none is for no read locks at all.
readLockTimeout	10000	An optional timeout, in milliseconds, for the read-lock, if supported by the read-lock. If the read-lock could not be granted and the timeout is triggered, then Camel skips the file. At the next poll Camel retries the file. Use a value of 0 or lower to indicate forever. Currently fileLock, changed and rename support the timeout. For FTP the default readLockTimeout value is 20000 instead of 10000. The readLockTimeout value must be higher than readLockCheckInterval, but a rule of thumb is to have a timeout that is at least twice the readLockCheckInterval. This ensures that ample time is allowed for the read lock process to try to grab the lock before the timeout.

Option	Default	Description
readLockCheckInterval	1000	Interval, in milliseconds, 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 changed read lock, you can set a higher interval period to cater for *slow writes*. The default of one second may be *too fast* if the producer is very slow writing the file. For FTP the default readLockCheckInterval is 5000.
readLockMinLength	1	This option applies only to readLock=changed. 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.
readLockLoggingLevel	WARN	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 changed, fileLock, rename.
readLockMarkerFile	true	Whether to use a marker file with the changed, rename, or exclusive read lock types. By default a marker file is used to guard against other processes picking up the same files. This behavior can be turned off by setting this option to false.
directoryMustExist	false	Similar to startingDirectoryMustExist but this applies during polling recursive sub directories.
doneFileName	null	If provided, Camel will only consume files if a *done* file exists. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders. The *done* file is **always** expected in the same folder as the original file..
exclusiveReadLock Strategy	null	Pluggable read-lock as a org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy implementation.

Option	Default	Description
maxMessagesPerPoll	0	An integer to define a maximum number of messages to gather per poll. By default no maximum is set. Can be used to set a limit of, for example, 1000 when starting up the server where there are thousands of files. Set a value of 0 or negative to disabled it. See more details at Batch Consumer. 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 maxMessagesPerPoll=500, then only the first 500 files will be picked up, and then sorted. You can use the eagerMaxMessagesPerPoll option and set this to false to allow to scan all files first and then sort afterwards.
eagerMaxMessagesPerPoll	true	Controls whether the limit from maxMessagesPerPoll is eager or not. If eager then the limit is applied during the scanning of files. Setting to false scans all files, and then performs sorting. Setting this option to false allows for sorting all files first, and then limiting the poll. Note that this requires a higher memory usage as all file details are in memory to perform the sorting.
minDepth	0	The minimum depth to start processing when recursively processing a directory. Using minDepth=1 means the base directory. Using minDepth=2 means the first sub directory.
maxDepth	Integer. MAX_VALUE	The maximum depth to traverse when recursively processing a directory.
processStrategy	null	A pluggable org.apache.camel.component.file.GenericFileProcessStrategy allowing you to implement your own readLock option or similar. Can also be used when special conditions must be met before a file can be consumed, such as when a special *ready* file exists. If this option is set then the readLock option does not apply.
startingDirectoryMustExist	false	Whether the starting directory must exist. Note that the autoCreate option is default-enabled, which means the starting directory is normally auto-created if it does not exist. You can disable autoCreate and enable this option to ensure that the starting directory must exist. Will throw an exception if the directory does not exist.

Option	Default	Description
pollStrategy	**null**	A pluggable org.apache.camel.spi.PollingConsumerPollStrategy allowing you to provide a custom implementation to control handling of errors occurring during the poll operation **before** an Exchange has been created and routed in Camel. In other words the error occurred while the polling was gathering information, for instance access to a file network failed so Camel cannot access it to scan for files. The default implementation logs the exception at WARN level and ignores it.
sendEmptyMessageWhenIdle	**false**	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.
consumer.bridgeErrorHandler	**false**	Bridges the consumer to the Camel routing Error Handler, which means that any exceptions occurring while trying to pickup files are processed as a message and handled by the routing Error Handler. By default the consumer uses the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that by default will be logged at WARN/ERROR level and ignored.
scheduledExecutorService	**null**	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.
scheduler	**null**	To use a custom scheduler to trigger the consumer to run.
backoffMultiplier	**0**	To let the scheduled polling consumer backoff if there has been a number of idles or errors in a row. The multiplier is then the number of polls to be skipped before the next actual attempt is made. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.
backoffIdleThreshold	**0**	The number of subsequent idle polls that should happen before the backoffMultipler is invoked.
backoffErrorThreshold	**0**	The number of subsequent error polls that should happen before the backoffMultipler is invoked.

Option	Default	Description
Producer only		
fileExist	Override	What to do if a file already exists with the same name. The following values can be specified: • Override, which is the default, replaces the existing file. • Append adds content to the existing file. • Fail throws a GenericFileOperationException, indicating that there is already an existing file. • Ignore silently ignores the problem and **does not** override the existing file, but assumes everything is satisfactory. • Move requires the corresponding moveExisting option to be configured. The option eagerDeleteTargetFile can be used to control what to do if moving a file when there already exists a file of that name, which would otherwise cause the move operation to fail. The Move option will move any existing files, before writing the target file. • TryRename is only applicable if tempFileName option is in use. This tries to rename the file from the temporary name to the actual name, without checking whether it already exists. This check may be faster on some file systems and especially FTP servers.
tempPrefix	null	This option is used to write the file using a temporary name and then, after the write is complete, rename it to the real name. Can be used to identify files being written and also to prevent consumers (not using exclusive read locks) reading in-progress files. It is often used by FTP when uploading big files.
tempFileName	null	The **same** as tempPrefix option but offering a more fine grained control on the naming of the temporary filename as it uses the File Language.
moveExisting	null	An Expression (such as File Language) used to compute the file name to use when fileExist=Move is configured. To move files into a backup subdirectory just enter backup. 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". Note 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 the current directory.

Option	Default	Description
`keepLastModified`	`false`	Keeps the last modified timestamp from the source file (if any). Will use the `Exchange.FILE_LAST_MODIFIED` header to located the timestamp. This header can contain either a `java.util.Date` or `long` with the timestamp. If the timestamp exists and the option is enabled it will set this timestamp on the written file. **Note:** This option only applies to the **file** producer. You *cannot* use this option with any of the ftp producers
`eagerDeleteTargetFile`	`true`	Whether or not to eagerly delete any existing target file. This option only applies when `fileExists=Override` and the `tempFileName` option are both set. You can use this to disable (set it to false) the deleting the target file before the temp file is written. For example you may write big files and want the target file to exist while the temp file is being written. This ensure the target file is not deleted until the very last moment, just before the temp file is being renamed to the target filename. This option is also used to control whether to delete any existing files when `fileExist=Move` is enabled, and an existing file exists. If the option `copyAndDeleteOnRenameFail` is `false`, then an exception will be thrown if an existing file existed. If it is `true`, then the existing file is deleted before the move operation.
`doneFileName`	`null`	If a filename is provided, Camel writes a second, empty, *done* file when the original file has been written. You can either specify a fixed filename, or use dynamic placeholders. The *done* file will **always** be written in the same folder as the original file.
`allowNullBody`	`false`	Used to specify if a null body is allowed during file writing. If set to `true` then an empty file will be created. If set to `false`, and attempting to send a null body to the file component, a `GenericFileWriteException` of 'Cannot write null body to file' will be thrown. If the `fileExist` option is set to `Override`, the file will be truncated, and if set to `Append,` the file will remain unchanged.
`forceWrites`	`true`	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 or audit logs. This yields better performance.
`Chmod`	`null`	Specify the file permissions sent by the producer. The chmod value must be between 000 and 777. If there is a leading digit (for example, as in 0755) it is ignored.

Message headers

The message headers shown in Table 6 can be used to affect the behavior of the file component

Table 6. File URI Message Headers

Header	Description
`org.apache.camel.file.name`	Specifies the output file name (relative to the endpoint directory) to be used for the output message when sending to the endpoint. If this is not present, a generated message ID is used instead.

JMS Component

The JMS component allows messages to be sent to (or consumed from) a JMS queue or topic. The JMS component uses Springs JMS support for declarative transactions, Spring's `JmsTemplate` for sending, and a `MessageListenerContainer` for consuming.

Endpoint URI format

JMS endpoints have the following URI format:

```
jms:[temp:][queue:|topic:]DestinationName[?Options]
```

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

```
jms:FOO.BAR
```

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

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

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

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

You can access temporary queues using the following URI format:

```
jms:temp:queue:DestinationName
```

Or temporary topics using the following URI format:

```
jms:temp:topic:DestinationName
```

This URI format enables multiple routes or processors or beans to refer to the same temporary destination. For example, you could create three temporary destinations and use them in routes as inputs or outputs by referring to them by name.

You can optionally add a list of query options, `?Options`, in the following format:

```
?Option=Value&Option=Value&Option=Value...
```

URI query options

JMS endpoints support the following URI query options. See http://camel.apache.org/jms.html for more details about how some of these options are used:

Table 7. JMS URI Query Options

Name	Default	Description
`acceptMessagesWhileStopping`	`false`	If `true`, a JMS consumer endpoint accepts messages while it is stopping.
`acknowledgementModeName`	`AUTO_ACKNOWLEDGE`	The JMS acknowledgement name, which is one of the following: `TRANSACTED`, `CLIENT_ACKNOWLEDGE`, `AUTO_ACKNOWLEDGE`, `DUPS_OK_ACKNOWLEDGE`.
`acknowledgementMode`	`-1`	The JMS acknowledgement mode, defined as an Integer. Allows you to set vendor-specific extensions to the acknowledgment mode. For the regular modes, set the `acknowledgementModeName` property instead.
`allowNullBody`	`true`	If `true`, allows sending messages with no body. If this option is `false` and the message body is null, then a `JMSException` is thrown.
`alwaysCopyMessage`	`False`	If `true`, the router 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 `replyToDestinationSelectorName` is set (the router automatically sets `alwaysCopyMessage` to `true` if a `replyToDestinationSelectorName` is set)

Name	Default	Description
asyncConsumer	false	Whether the JmsConsumer processes the Exchange asynchronously. If this is enabled, the JmsConsumer may pick up 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 not be processed strictly in order. If false (as default) then the Exchange is fully processed before the JmsConsumer picks up the next message from the JMS queue. Note if transacted has been enabled, then asyncConsumer=true does not run asynchronously, as transactions must be executed synchronously.
asyncStartListener	false	Whether to start the JmsConsumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, it may block while retrying. This causes Camel to block while starting routes. Setting this option to true allows routes to startup, while the JmsConsumer connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, note that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages. You can then restart the route to retry.
asyncStopListener	false	Whether to stop the JmsConsumer message listener asynchronously, when stopping a route.
autoStartup	true	If true, the consumer container starts up automatically.
cacheLevel	-1	Sets the cache level ID for the underlying JMS resources.
cacheLevelName	CACHE_AUTO	Sets the cache level by name for the underlying JMS resources. Possible values are: CACHE_AUTO, CACHE_CONNECTION, CACHE_CONSUMER, CACHE_NONE, and CACHE_SESSION.

Name	Default	Description
`clientId`	`null`	Sets the JMS client ID. This value must be unique and can only be used by a single JMS connection instance. It is typically required only for *durable* topic subscriptions. You may prefer to use *virtual topics* instead.
`consumerType`	`Default`	The consumer type determines which Spring JMS listener should be used. This option can have one of the following values: • `Default`—for `DefaultMessageListenerContainer`. • `Simple`—for `SimpleMessageListenerContainer`. • `ServerSessionPool`—for `serversession.ServerSessionMessageListenerContainer`. Where each of these classes belongs to the `org.springframework.jms.listener` Java package. If you set `useVersion102=true`, the router will use the corresponding JMS 1.0.2 Spring classes instead.
`concurrentConsumers`	`1`	Specifies the default number of concurrent consumers.
`connectionFactory`	`null`	The default JMS connection factory to use for the `listenerConnectionFactory` and `templateConnectionFactory`, if neither are specified.

Name	Default	Description
defaultTaskExecutorType	null	Specifies what default TaskExecutor type to use in the DefaultMessageListenerContainer, both for consumer endpoints and the ReplyTo consumer of producer endpoints. Possible values are: • SimpleAsync (uses Spring's SimpleAsyncTaskExecutor) • ThreadPool (uses Spring's ThreadPoolTaskExecutor with optimal values - cached threadpool-like). If not set, it defaults to using a cached ThreadPool for consumer endpoints and SimpleAsync for reply consumers. The use of ThreadPool is recommended to reduce "thread trash" in elastic configurations with dynamically increasing and decreasing concurrent consumers.
deliveryMode	null	Specifies the delivery mode to be used. Possible values are those defined by javax.jms.DeliveryMode.
deliveryPersistent	true	Is persistent delivery 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 org.springframework.jms.support.destination.DestinationResolver that allows you to use your own resolver (for example, to look up the real destination in a JNDI registry).
disableReplyTo	false	Do you want to ignore the JMSReplyTo header and so treat messages as InOnly by default and not send a reply back?

Name	Default	Description
disableTimeToLive	false	Use this option to disable the time to live value on sent messages. For example, in a request/reply over JMS, Camel by default uses the requestTimeout value as time to live on the message being sent. However the sender and receiver systems have to have their clocks synchronized, which is not always the case. You can use disableTimeToLive=true to **not** set a time to live value on the sent message. Then the message will not expire on the receiver system.
durableSubscriptionName	null	The durable subscriber name for specifying durable topic subscriptions.
eagerLoadingOfProperties	false	Enables eager loading of JMS properties as soon as a message is received. This feature is generally inefficient, because the JMS properties might not be required. But eager loading can be useful for testing purpose, to ensure JMS properties can be understood and handled correctly.
errorHandler	null	Specifies a org.springframework.util.ErrorHandler to be invoked in case of any uncaught exceptions thrown while processing a Message. By default these exceptions will be logged at the WARN level, if no errorHandler has been configured. You can configure the logging level and whether stack traces should be logged using the next two options.
errorHandlerLoggingLevel	WARN	Configure the default errorHandler logging level for logging uncaught exceptions.
errorHandlerLogStackTrace	true	Controls whether stacktraces should be logged by the default errorHandler or not.
exceptionListener	null	The JMS Exception Listener used to be notified of any underlying JMS exceptions.
explicitQosEnabled	false	If true, the properties, deliveryMode, priority, and timeToLive, are used when sending messages.

Name	Default	Description
exposeListenerSession	true	If true, the listener session is exposed when consuming messages.
forceSendOriginalMessage	false	When using mapJmsMessage=false Camel creates 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 true to force Camel to send the original JMS message that was received.
idleConsumerLimit	1	Specify a limit for the number of consumers that are allowed to be idle at any given time.
idleTaskExecutionLimit	1	Specify the limit for idle executions of a receive task, not having received any message within its execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in the case of dynamic scheduling; see the maxConcurrentConsumers setting).
includeSentJMSMessageID	false	This is only applicable when sending to a JMS destination using InOnly (for example, "fire and forget"). Enabling this option enriches the Camel Exchange with the actual JMSMessageID that was used by the JMS client when the message was sent to the JMS destination.
includeAllJMSXProperties	false	Whether to include all JMSXxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSXAppID, and JMSXUserID etc. **Note:** If you are using a custom headerFilterStrategy then this option does not apply.
jmsMessageType	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..

Name	**Default**	**Description**
`jmsKeyFormatStrategy`	`default`	A 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: • 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 accept illegal characters in JMS header keys. You can provide your own implementation of the `org.apache.camel.component.jms.JmsKeyFormatStrategy` and refer to it using the # notation.
`jmsOperations`	`null`	Enables you to use your own implementation of the `org.springframework.jms.core.JmsOperations` interface. The router uses the `JmsTemplate` class by default. Can be used for testing purpose.
`lazyCreateTransactionManager`	`true`	If `true`, Camel creates a `JmsTransactionManager`, if there is no `transactionManager` injected when option `transacted=true`.
`listenerConnectionFactory`	`null`	The JMS connection factory used for consuming messages.
`mapJmsMessage`	`true`	If `true`, Camel automatically maps the received JMS message to an appropriate payload type, such as `javax.jms.TextMessage` to a `String`
`maximumBrowseSize`	`-1`	Limits the number of messages fetched when browsing endpoints using Browse or the JMX API.
`maxConcurrentConsumers`	`1`	Specifies the maximum number of concurrent consumers.
`maxMessagesPerTask`	`1`	The number of messages per task.
`messageConverter`	`null`	The Spring Message Converter.
`messageIdEnabled`	`true`	If `true`, message IDs are added to sent messages.

Name	Default	Description
messageListenerContainer FactoryRef	null	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	Should timestamps be enabled by default on sending messages.
password	null	The password for the connector factory.
priority	-1	Values of > 1 specify the message priority when sending, if the explicitQosEnabled property is specified.
preserveMessageQos	false	Set to true, if you want to send message using the QoS settings specified on the message, instead of the QoS settings on the JMS endpoint.
pubSubNoLocal	false	Specifies whether to inhibit the delivery of messages published by its own connection.
selector	null	Sets the JMS Selector which is an SQL 92 predicate used to apply to messages to filter them at the message broker. You may have to encode special characters such as = as %3D.
receiveTimeout	none	The timeout when receiving messages.
recoveryInterval	none	The recovery interval.
replyTo	null	Provides an explicit ReplyTo destination, which overrides any incoming value of Message.getJMSReplyTo().

Name	Default	Description
replyToCacheLevelName	CACHE_CONSUMER	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 by default uses: • CACHE_CONSUMER for exclusive or shared w/ replyToSelectorName. • CACHE_SESSION for shared without replyToSelectorName. Some JMS brokers such as IBM WebSphere may need to set replyToCacheLevelName=CACHE_NONE in order 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.
replyToOverride	null	Provides an explicit ReplyTo destination in the JMS message, which overrides the setting of replyTo. It is useful if you want to forward the message to a remote Queue and receive the reply message from the ReplyTo destination.
replyToTempDestinationAffinity	endpoint	Specifies how temporary queues are used for the replyTo destination sharing strategy. This option can take one of the following values: • component—a single temporary queue is shared among all producers for a given component instance. • endpoint—a single temporary queue is shared among all producers for a given endpoint instance. • producer—a single temporary queue is created for each producer.
replyToDestination	null	Provides an explicit replyTo destination which overrides any incoming value of Message.getJMSReplyTo().
replyToDestinationSelectorName	null	When using a shared queue (that is, not using a temporary reply queue), this option sets the name of a JMS selector that is used to filter replies.
replyToDeliveryPersistent	true	Specifies whether persistent delivery is used by default for replies.

Name	**Default**	**Description**
`replyToType`	`null`	Specifies the kind of strategy to use for replyTo queues when doing request/reply over JMS. Possible values are: `Temporary`, `Shared`, or `Exclusive`. By default Camel will use temporary queues. However if `replyTo` has been configured, then `Shared` is used by default. This option allows you to use exclusive queues instead of shared ones. Note that `Shared` reply queues have lower performance than `Temporary` and `Exclusive`.
`requestTimeout`	`20000`	The timeout when sending messages.
`requestTimeoutCheckerInterval`	`1000`	Configures how often Camel should check for timed out Exchanges when doing request/reply over JMS. By default Camel checks once per second. Lower this interval, to check more frequently in order to react faster when a timeout occurs. The timeout is determined by the option *requestTimeout*.
`selector`	`null`	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, for example "`=`" as "`%3D`".
`serverSessionFactory`	`null`	The JMS ServerSessionFactory if you wish to use ServerSessionFactory for consumption.
`subscriptionDurable`	`false`	Enabled by default if you specify a durableSubscriberName and a clientId.
`taskExecutor`	`null`	Allows you to specify a custom task executor for consuming messages.
`taskExecutorSpring2`	`null`	Specifies a custom task executor for consuming messages when using Spring 2.x with Camel.
`templateConnectionFactory`	`null`	The JMS connection factory used for sending messages.

Name	**Default**	**Description**
`testConnectionOnStartup`	`false`	Specifies whether to test the connection on startup. This ensures that when Camel starts 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.
`timeToLive`	`null`	Is a time to live specified when sending messages?
`transacted`	`false`	Specifies whether transacted mode is used for sending/receiving messages.
`transactedInOut`	`false`	Specifies whether transacted mode is used with the *InOut* exchange pattern.
`transactionManager`	`null`	The Spring transaction manager to use.
`transactionName`	`null`	The name of the transaction to use.
`transactionTimeout`	`null`	The timeout value of the transaction if using transacted mode.
`transferException`	`false`	If this is enabled, when you are using Request Reply messaging (InOut) and an Exchange fails on the consumer side, then the caused `Exception` is sent back in response as a `javax.jms.ObjectMessage`. 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. Note 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 `org.apache.camel.RuntimeCamelException` when returned to the producer.

Name	Default	Description
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 the exclusion at WARN level. You **must** enable this option on both the producer and consumer side, so Camel knows the payload is an Exchange and not a regular payload.
username	null	The username for the connector factory.
useMessageIDAsCorrelationID	false	Specifies whether `JMSMessageID` is used as the `JMSCorrelationID` for *InOut* messages. By default, the router uses a GUID.
useVersion102	-	No longer supported.

Configuring in XML

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

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

<bean id="activemq" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL"
value="vm://localhost?broker.persistent=false"/>
    </bean>
  </property>
</bean>
```

You can configure as many JMS component instances as you wish and give them a unique name using the `id` attribute. The preceding example creates an `activemq` component. You could take a similar approach to configuring MQSeries, 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, given the component name, `activemq`, you can then refer to destinations as `activemq:[queue:|topic:]`*`DestinationName`*. 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 connection factory

If you are using a J2EE container, you might want to look up JNDI to find your `ConnectionFactory` rather than use the usual `<bean>` mechanism in spring. You can do this using Spring's factory bean or the new XML namespace. For example:

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

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

Enabling transactions

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

```
?transacted=true&transactionManager=TranssactionManager
```

Where the *`TransactionManager`* is typically the `JmsTransactionManager`.

Durable subscriptions

If you wish to use durable topic subscriptions, you need to specify both the `clientId` and `durableSubscriberName` query options. Note that the value of the `clientId` must be unique and can only be used by a single JMS connection instance in your entire network. You may prefer to use Virtual Topics instead to avoid this limitation. For more background, see Durable Messaging at `http://activemq.apache.org/how-do-durable-queues-and-topics-work.html`.

Adding message headers

When using message headers; the JMS specification states that header names must be valid Java identifiers. So, by default, the JMS component will ignore any headers which do not match this rule. Try to name your headers as if they are valid Java identifiers. One benefit of this is that you can then use your headers inside a JMS Selector (whose SQL92 syntax mandates headers in the form of Java identifiers).

Cache settings

If you are using XA or running in a J2EE container, you might need to set the `cacheLevelName` to be `CACHE_NONE`. We have found it necessary to disable caching with JBoss and JTA/XA.

Using the JMS component with ActiveMQ

The JMS component exploits Spring 2's `JmsTemplate` for sending messages. This is not ideal for use in a non-J2EE container and typically requires a caching JMS provider to avoid poor performance. So, if you intend to use Apache ActiveMQ (see `http://activemq.apache.org/`) as your Message Broker, we recommend that you either:

- Use the ActiveMQ component, which is already configured to use ActiveMQ efficiently, or
- Use the `PoolingConnectionFactory` in ActiveMQ.

SOAP

The SOAP protocol does not have a dedicated component. It is supported through the CXF component—see CXF Component.

Websphere MQ Component

The Websphere MQ component is a specialized JMS component that is used to integrate IBM's Websphere MQ into the Artix Java router. Because the Websphere MQ component is derived from the JMS component, all of the properties provided by the JMS component are also available to the Websphere MQ component. In addition, the Websphere MQ component automatically configures the underlying IBM connection factory for you.

NOTE: You must have a license for the Websphere MQ product to use this component. The required Websphere libraries are *not* provided with Artix.

Adding the MQ component

There is no need to add the Websphere MQ component to the Camel context; it is automatically loaded by the router core.

Endpoint URI format

The Websphere MQ component has a URI format that is almost identical to the JMS URI format, except that the `jms:` prefix is replaced by `mq:`.

```
mq:[temp:][queue:|topic:]DestinationName[?Options]
```

For a detailed description of the analogous JMS URI format, see Endpoint URI format.

URI query options

MQ endpoints support all of the JMS query options—see Table 7. In addition, the MQ endpoints also support the following query options:

Table 8. MQ URI Query Options

Name	Default	Description
`userName`	`null`	User name for the Websphere MQ connection.
`userPassword`	`null`	User password for the Websphere MQ connection.
`explicitQosEnabled`	`true`	Same as the corresponding JMS option, with different default. *The value of this option has been optimized for Websphere MQ. Do not change!*
`messageIdEnabled`	`true`	Same as the corresponding JMS option. *The value of this option has been optimized for Websphere MQ. Do not change!*
`replyToDeliveryPersistent`	`false`	Same as the corresponding JMS option, with different default. *The value of this option has been optimized for Websphere MQ. Do not change!*
`useMessageIDAsCorrelationID`	`true`	Same as the corresponding JMS option, with different default. *The value of this option has been optimized for Websphere MQ. Do not change!*

Demonstration code with transaction propagation

In the Artix samples, there is an advanced demonstration that shows how to configure the Java router to act as a bridge between FUSE Message Broker (Apache ActiveMQ) and Websphere MQ, with full support for XA transaction propagation. The demonstration code can be found at the following location:

```
ArtixRoot/java/samples/transports/jms/mqi_bridge
```

And the router configuration can be found in the following files:

```
mqi_bridge/src/bridge/com/iona/bridge/routes.xml
mqi_bridge/src/bridge/com/iona/bridge/components.xml
```