PROGRESS SOFTWARE

Artix ESB

Java Router, Deployment Guide

Version 5.5
December 2008

Java Router, Deployment Guide

Progress Software

Version 5.5

Published 19 Dec 2008

Legal Notices

Table of Contents

List of Figures

List of Tables

List of Examples

Deploying a Standalone Router

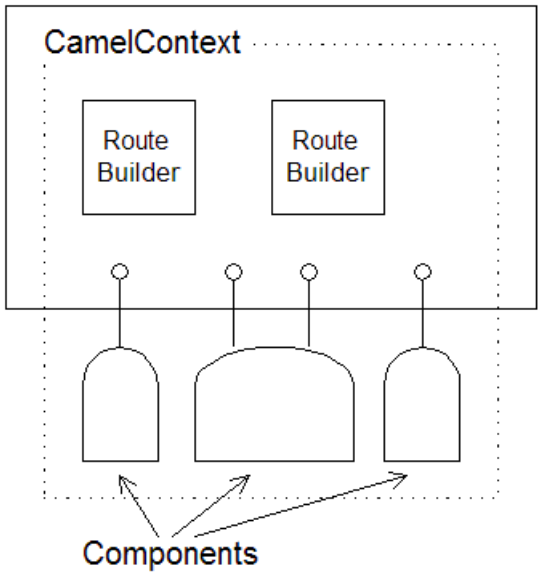
This chapter describes how to deploy the Java Router in standalone mode. This means that you can deploy the router independent of any container, but some extra programming steps are required.

Introduction to Standalone Deployment

Overview

Figure 1 on page 12 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

Overview

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 14 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 on page 16).

❹ The second step in initializing the Camel context is to add one or more RouteBuilder objects (see Adding RouteBuilders to the Camel Context on page 18).

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 on page 16 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 ActiveMQCon
nectionFactory("vm://localhost?broker.persistent=false"); ❶
        context.addComponent("test-jms", JmsComponent.jmsCom
ponentAutoAcknowledge(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 componenet 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

Overview

`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 on page 18 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 on page 16.

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). See for more details about implementing custom processors.

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

Running a Standalone Application

Setting the CLASSPATH

Configure your application's CLASSPATH as follows:

1. Add all of the JAR files in *ArtixRoot*`/java/lib/camel/1.5.1.0-fuse` to your CLASSPATH. This step can be simplified if you use a general-purpose build tool such as Apache Maven [http://maven.apache.org/] or Apache Ant [http://ant.apache.org/] to build your application.

Running the application

Assuming that you have coded a `main()` method, as described in Defining a Standalone Main Method on page 14, 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 typically 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

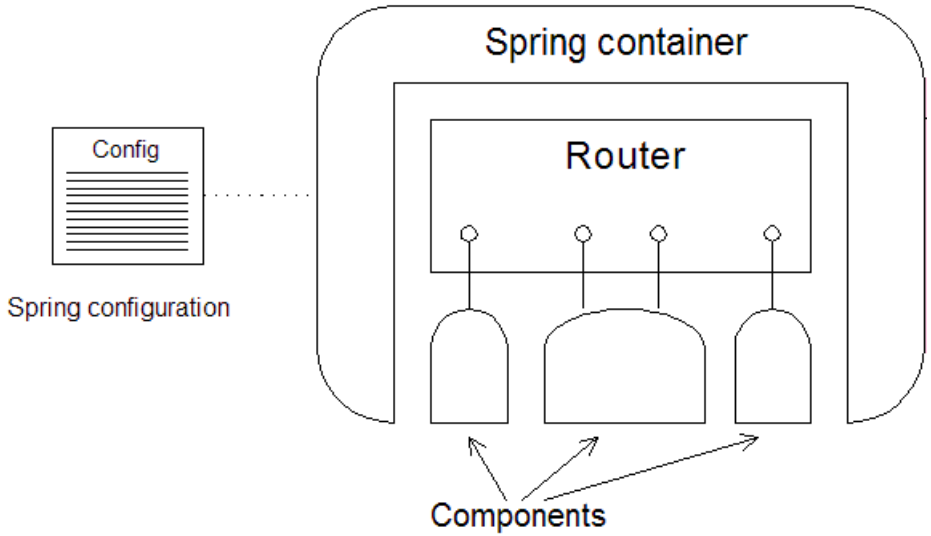
This chapter describes how to deploy the Java Router into a Spring container. A notable feature of the Spring container deployment is that it enables you to specify routing rules in an XML configuration file.

Introduction to Spring Deployment

Overview

Figure 2 on page 22 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 on page 25).

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 on page 26).

Defining a Spring Main Method

Overview

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 on page 24 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 options

Spring Configuration

Overview

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 of course 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 on page 25 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 ht
tp://www.springframework.org/schema/beans/spring-beans-2.0.xsd
 ❶
       http://activemq.apache.org/camel/schema/spring ht
tp://activemq.apache.org/camel/schema/spring/camel-spring.xsd">
 ❷

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

```
  </camelContext>
</beans>
```

Where the preceding configuration can be explained as follows:

❶ This line specifies the location of the Spring framework schema. The URL should represent a real, physical location from where you can download the schema. The version of the Spring schema currenlty supported by Java Router is Spring 2.0.

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://activemq.apache.org/camel/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, it is relatively straightforward to figure what properties you can set by consulting the JavaDoc for the relevant component.

Example 6 on page 26 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://act
ivemq.apache.org/camel/schema/spring">
    <!-- Java packages (not shown) ... -->
  </camelContext>
```

```
  <!-- Configure the default ActiveMQ broker URL -->
  <bean id="jms" class="org.apache.camel.component.jms.JmsCom
ponent"> ❶
    <property name="connectionFactory"> ❷
      <bean class="org.apache.activemq.ActiveMQConnectionFact
ory"> 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* on page 29.

Running a Spring Application

Setting the CLASSPATH

Configure your application's CLASSPATH as follows:

1. Add all of the JAR files in *ArtixRoot*`/java/lib/camel/1.5.1.0-fuse` to your CLASSPATH. This step can be simplified if you use a general-purpose build tool such as Apache Maven [http://maven.apache.org/] or Apache Ant [http://ant.apache.org/] to build your application.
2. Add the directory containing `META-INF/spring/camel-context.xml` to your CLASSPATH. For example, if your Spring configuration file is `/var/my_router_app/META-INF/spring/camel-context.xml`, you would add the following directory to your CLASSPATH:

```
/var/my_router_app
```

Running the application

Assuming that you have coded a `main()` method, as described in Defining a Spring Main Method on page 24, 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 typically 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

Overview

The CORBA protocol does not have a dedicated component. It is supported through the CXF component—see CXF Component on page 31.

CXF Component

Introduction to CXF Component

Overview

The CXF component enables you to access endpoints using the Apache CXF [http://incubator.apache.org/cxf/] 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
- XML

And CXF supports the following transports:

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

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 on page 34.
- Bean Endpoint URI on page 36.

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 on page 34.

Table 1. CXF URI Query Options

Option	Description
`address`	The endpoint address (overriding the value that appears in the fist part of theCXF URI).
`dataFormat`	The format used to represent messages internally. Currently, the only supported format is POJO (Plain Old Java Object).
`serviceClass`	A service endpoint interface (SEI) class name. If the SEI class is appropriately annotated, 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).
`wsdlURL`	Location of the WSDL contract file (defaults to the value of the annotation in the service class, if one is specified).

You can combine these options in various ways, in order to provide the requisite details about a service endpoint. For example, you would typically define a CXF URI in one of the following ways:

- *CXF URI based on an SEI class*—if you specify just the `serviceClass` option, CXF implicitly takes the port name, service name, and WSDL location from the annotations on the SEI class.

- *CXF URI with explicit options*—alternatively, you can specify the port name, `portName`, service name, `serviceName`, and WSDL location, `wsdlURL`, expicitly using the CXF query options.

Bean Endpoint URI

Endpoint URI format

The CXF bean endpoint URI conforms to the following format:

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

Where *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://act
ivemq.apache.org/camel/schema/cxfEndpoint"
       ...>
    ...
    <cxf:cxfEndpoint id="BeanID"
        serviceClass="serviceClassName"
        address="https://localhost:58001/GreeterService/Basi
cAuthPort"
        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 on page 34 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.

Attribute	Description
`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.
`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 on page 38.
`cxf:handlers`	A JAX-WS handler list for the JAX-WS endpoint. See JAX-WS Configuration [http://cwiki.apache.org/CXF20DOC/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.

Child Element	Description
`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 on page 34. 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

Overview

A CXF endpoint allows you to select different data formats for the propagated messages, as shown in Table 4 on page 39. 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 consists of a `java.util.List` containing the Java parasmeters to the method being invoked on the target server.
`PAYLOAD`	The message body contains the contents of the `soap:body` element after the endpoint's message configuration has been applied.
`MESSAGE`	The message body contains the raw message that is received from the transport layer.

Accessing a message in POJO data format

The POJO data format is based on the CXF invoker [http://cwiki.apache.org/CXF20DOC/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)ex
change.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)ex
change.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.Unknown
PersonFault();
            personFault.setPersonId("");
            org.apache.camel.wsdl_first.UnknownPersonFault
fault =
               new org.apache.camel.wsdl_first.UnknownPerson
Fault("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 re
turn 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 = Cas
tUtils.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

You can use `Header.HEADER_LIST` as the key to set or get the SOAP headers and use the `List<Element>` type to set or get SOAP body elements. For example:

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

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

       List<SoapHeader> headers = CastUtils.cast((List<?>)mes
sage.getMessage().get(Header.HEADER_LIST));
        assertNotNull("We should get the headers here", head
ers);
        assertEquals("Get the wrong headers size", head
ers.size(), 1);
        assertEquals("Get the wrong namespace URI" , ((Ele
ment)(headers.get(0).getObject())).getNamespaceURI(), "ht
tp://camel.apache.org/pizza/types");

    }

})
.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 `MESSAGE` 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).throwFault(SOAP_FAULT);
```

If your CXF endpoint is configured to use the MESSAGE data format, you could set the the SOAP Fault message in the message body and set the response code in the message header. 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.RE
SPONSE_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 ArrayL
ist<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_CON
TEXT , 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 = Cas
tUtils.cast((Map)out.getHeader(Client.RESPONSE_CONTEXT));
        assertNotNull(responseContext);
        assertEquals("Get the wrong wsdl opertion name",
"{http://apache.org/hello_world_soap_http}greetMe", respon
seContext.get("javax.xml.ws.wsdl.operation").toString());
```

File Component

Overview

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 on page 44.

Table 5. File URI Query Options

Option	Default	Description
`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.
`recursive`	`true`	If `true` and the file URI specifies a directory path, the file component polls for changes in all sub-directories.
`lock`	`true`	If `true`, lock the file for the duration of the processing.
`regexPattern`	`null`	Only process files that match the regular expression pattern.
`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.
`moveNamePrefix`	`null`	Specifies the string to prepend to the file's path name when moving it. For example to move processed files into the `done` directory, set this option to `done/`.

Option	Default	Description
`moveNamePostfix`	`null`	Specifies the string to append to the file's path name when moving it. For example to rename processed files from `foo` to `foo.old` set this value to `.old`.
`append`	`true`	When writing to a file, if this option is `true`, append to the end of the file; if this option is `false`, replace the file.

Message headers

The message headers shown in Table 6 on page 45 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

Overview

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:

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.
`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)
`autoStartup`	`true`	If `true`, the consumer container starts up automatically.
`cacheLevel`	`-1`	Sets the cache level ID for the underlying JMS resources.
`cacheLevelName`	`CACHE_CONNECTION` (but when SPR-3890 is fixed, it will be `CACHE_CONSUMER`).	Sets the cache level name for the underlying JMS resources.
`clientId`	`null`	Sets the JMS client ID. This value must be unique and can only be used by a single JMS connection

Name	Default	Description
		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.
`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
`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
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.
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.
exposeListenerSession	true	If true, the listener session is exposed when consuming messages.
idleTaskExecutionLimit	1	Specify the limit for idle executions of a receive task, not having received any message within its execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in the case of dynamic scheduling; see the maxConcurrentConsumers setting).
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.
listenerConnectionFactory	null	The JMS connection factory used for consuming messages.
maxConcurrentConsumers	1	Specifies the maximum number of concurrent consumers.
maxMessagesPerTask	1	The number of messages per task.
messageConverter	null	The Spring Message Converter.
messageIdEnabled	true	If true, message IDs are added to sent messages.
messageTimestampEnabled	true	Should timestamps be enabled by default on sending messages.

Name	Default	Description
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.
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.

Name	Default	Description
replyToDeliveryPersistent	true	Specifies whether persistent delivery is used by default for replies.
requestTimeout	20000	The timeout when sending messages.
serverSessionFactory	null	The JMS ServerSessionFactory if you wish to use ServerSessionFactory for consumption.
subscriptionDurable	false	Enabled by default if you specify a durableSubscriberName and a clientId.
taskExecutor	null	Allows you to specify a custom task executor for consuming messages.
templateConnectionFactory	null	The JMS connection factory used for sending messages.
timeToLive	null	Is a time to live specified when sending messages.
transacted	false	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.
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	false	Should the old JMS API be used.

Configuring in XML

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

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

<bean id="activemq" class="org.apache.camel.component.jms.Jm
```

```
sComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFact
ory">
      <property name="brokerURL" value="vm://local
host?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, TibCo, BEA, Sonic, and so on.

Once you have a named JMS component you can then refer to endpoints within that component using URIs. For example, 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 lookup 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.Jm
sComponent">
  <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 [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 with TibCo EMS 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 [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

Overview

The SOAP protocol does not have a dedicated component. It is supported through the CXF component—see CXF Component on page 31.

Websphere MQ Component

Overview

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 on page 46.

URI query options

MQ endpoints support all of the JMS query options—see Table 7 on page 47. 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!*

Name	Default	Description
`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
```