Artix ESB

Java Router, Deployment Guide

Version 5.1
December 2007

Making Software Work Together™

Java Router, Deployment Guide

IONA Technologies

Version 5.1

Published 19 Dec 2007

Trademark and Disclaimer Notice

Copyright Notice

Table of Contents

Preface **11**
Document Conventions 12
Deploying a Standalone Router **15**
Introduction to Standalone Deployment 16
Defining a Standalone Main Method 18
Adding Components to the Camel Context 20
Adding RouteBuilders to the Camel Context 22
Running a Standalone Application 24
Deploying into a Spring Container **25**
Introduction to Spring Deployment 26
Defining a Spring Main Method 28
Spring Configuration 29
Running a Spring Application 32
Components **33**
CORBA 34
CXF Component 35
File Component 37
SOAP 39

List of Figures

1. Standalone Router 16
2. Router Deployed in a Spring Container 26

List of Tables

1. CXF URI Query Options 36
2. File URI Query Options 37
3. File URI Message Headers 38

List of Examples

1. Standalone Main Method 18
2. Adding a Component to the Camel Context 20
3. Adding a RouteBuilder to the Camel Context 22
4. Spring Main Method 28
5. Basic Spring XML Configuration 29
6. Configuring Components in Spring 30

Preface

Table of Contents

Document Conventions 12

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`fixed width`	Fixed width (Courier 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).

Admonition conventions

This book uses the following conventions for admonitions:

	Notes display information that may be useful, but not critical.
	Tips provide hints about completing a task or using a tool. They may also provide information about workarounds to possible problems.
	Important notes display information that is critical to the task at hand.
	Cautions display information about likely errors that can be encountered. These errors are unlikely to cause damage to your data or your systems.
	Warnings display information about errors that may cause damage to your systems. Possible damage from these errors include system failures and loss of data.

Deploying a Standalone Router

Summary

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.

Table of Contents

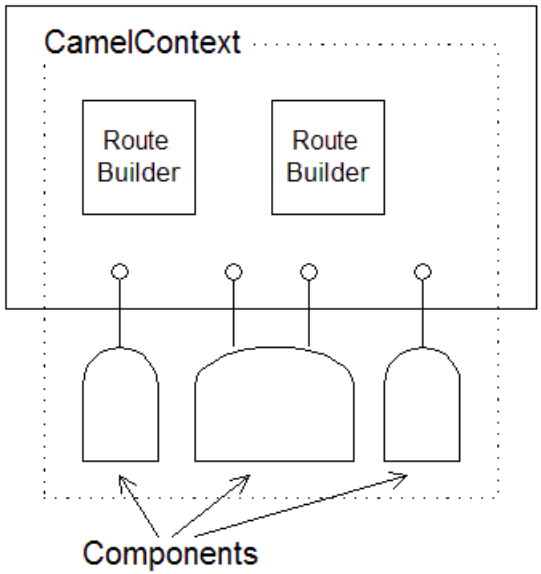
Introduction to Standalone Deployment 16
Defining a Standalone Main Method 18
Adding Components to the Camel Context 20
Adding RouteBuilders to the Camel Context 22
Running a Standalone Application 24

Introduction to Standalone Deployment

Overview

Figure 1, “Standalone Router” 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, "Standalone Main Method" 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, "Adding a Component to the Camel Context" 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 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, "Adding a RouteBuilder to the Camel Context" 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, “Adding a Component to the Camel Context”.

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 *`RouterRoot`*`/lib` and *`RouterRoot`*`/lib/optional` 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, 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

Summary

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.

Table of Contents

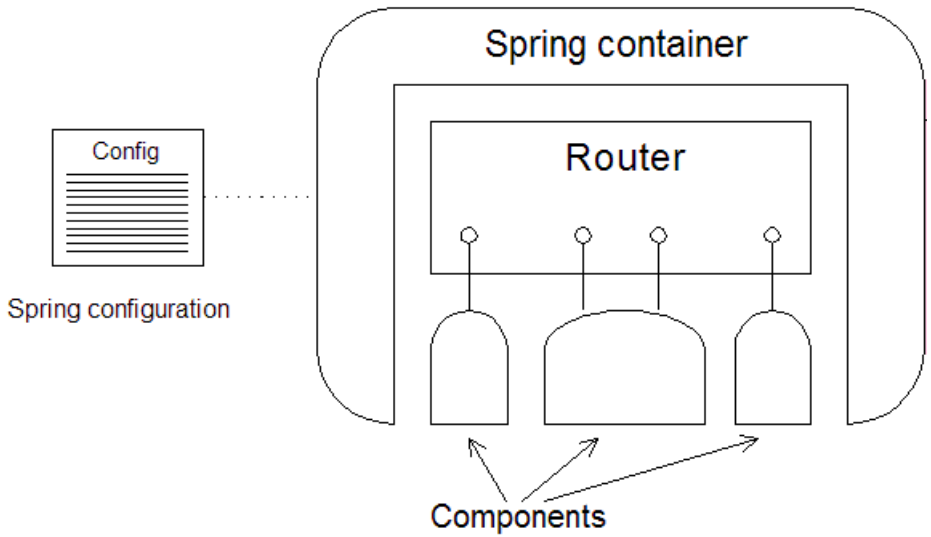
Introduction to Spring Deployment 26
Defining a Spring Main Method 28
Spring Configuration 29
Running a Spring Application 32

Introduction to Spring Deployment

Overview

Figure 2, “Router Deployed in a Spring Container” 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

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, “Spring Main Method” 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, “Basic Spring XML Configuration” 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-2.0.xsd
 ❶
       http://activemq.apache.org/camel/schema/spring
http://activemq.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>
```

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, “Configuring Components in Spring” 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://activemq.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.JmsComponent"> ❶
    <property name="connectionFactory"> ❷
       <bean
class="org.apache.activemq.ActiveMQConnectionFactory"> ❸
         <property name="brokerURL"
value="vm://localhost?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

Setting the CLASSPATH

Configure your application's CLASSPATH as follows:

1. Add all of the JAR files in *`RouterRoot`*`/lib` and *`RouterRoot`*`/lib/optional` 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.

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 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

Summary

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.

Table of Contents

CORBA 34
CXF Component 35
File Component 37
SOAP 39

CORBA

Overview

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

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.

Endpoint URI format

A CXF endpoint has a URI that 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, “CXF URI Query Options”.

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.

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 2, “File URI Query Options”.

Table 2. 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 , 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 3, "File URI Message Headers" can be used to affect the behavior of the file component.

Table 3. 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.

SOAP

Overview

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