

# FUSE Mediation Router

## Deployment Guide

Version 1.4  
September 2008

# Deployment Guide

Progress Software

Version 1.4

Published 14 Oct 2008

Copyright © 2001-2008 IONA Technologies PLC

## ***Trademark and Disclaimer Notice***

IONA Technologies PLC and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from IONA Technologies PLC, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

IONA, IONA Technologies, the IONA logo, Orbix, High Performance Integration, Artix, FUSE, and Making Software Work Together are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

## ***Copyright Notice***

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice. Portions of this document may include Apache Foundation documentation, all rights reserved.

# Table of Contents

<b>Deploying a Standalone Router</b> .....	<b>9</b>
Introduction to Standalone Deployment .....	10
Defining a Standalone Main Method .....	12
Adding Components to the Camel Context .....	14
Adding RouteBuilders to the Camel Context .....	16
Running a Standalone Application .....	18
<b>Deploying into a Spring Container</b> .....	<b>19</b>
Introduction to Spring Deployment .....	20
Defining a Spring Main Method .....	22
Spring Configuration .....	23
Running a Spring Application .....	26



# List of Figures

1. Standalone Router .....	10
2. Router Deployed in a Spring Container .....	20



# List of Examples

1. Standalone Main Method .....	12
2. Adding a Component to the Camel Context .....	14
3. Adding a RouteBuilder to the Camel Context .....	16
4. Spring Main Method .....	22
5. Basic Spring XML Configuration .....	23
6. Configuring Components in Spring .....	24



# Deploying a Standalone Router

*This chapter describes how to deploy the Mediation 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 .....	10
Defining a Standalone Main Method .....	12
Adding Components to the Camel Context .....	14
Adding RouteBuilders to the Camel Context .....	16
Running a Standalone Application .....	18

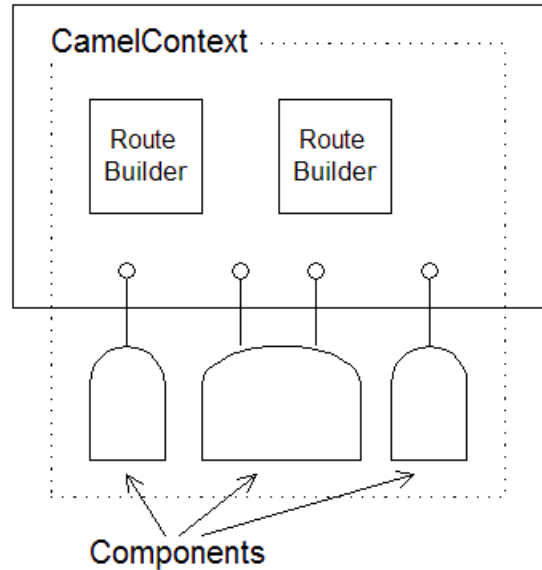
# Introduction to Standalone Deployment

---

## Overview

Figure 1 on page 10 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 12](#) 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 you need for your routes (see [Adding Components to the Camel Context on page 14](#)).
- 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 16](#)).
- 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 14](#) 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 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

### 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 16](#) 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:

- 1 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.



## Note

When defining routes that include transactional components, you must inherit from the class, `org.apache.camel.spring.SpringRouteBuilder`, instead of `RouteBuilder`. See [Transactional Client](#) in the *Implementing Enterprise Integration Patterns*.

- 2 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 14](#).
- 3 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.
- 4 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/) [http://maven.apache.org/] or [Apache Ant](http://ant.apache.org/) [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 12](#), 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/) [http://www.eclipse.org/] or [IntelliJ](http://www.jetbrains.com/idea/) [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 Mediation 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 .....	20
Defining a Spring Main Method .....	22
Spring Configuration .....	23
Running a Spring Application .....	26

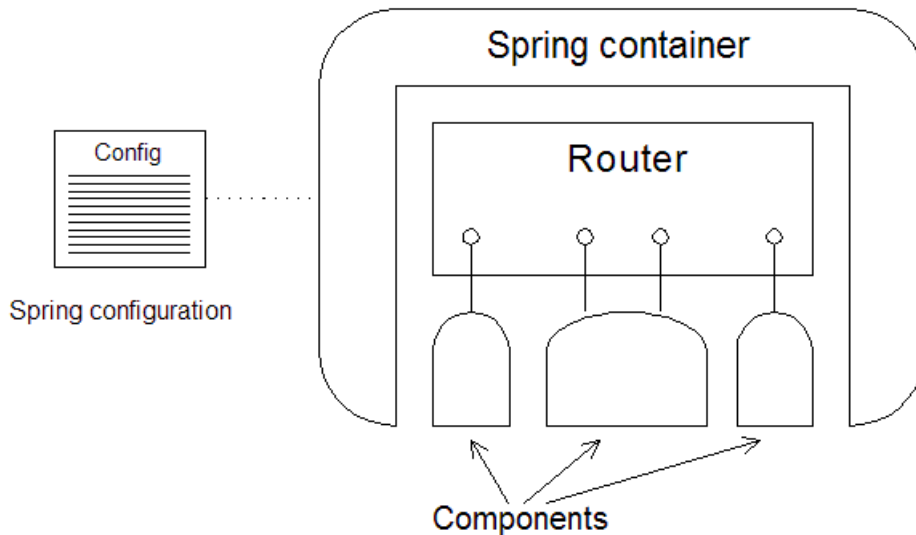
## Introduction to Spring Deployment

---

### Overview

Figure 2 on page 20 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, Mediation 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 23](#)).

---

### 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, Mediation 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 24](#)).

## Defining a Spring Main Method

---

### Overview

Mediation 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 22](#) 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

---

## 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 in the following directory, relative to your CLASSPATH (that is, the parent of `META-INF` must appear on your CLASSPATH):

```
META-INF/spring/
```

The Spring container reads *any* file that matches the pattern, `META-INF/spring/*.xml` and there can be more than one such file. For the examples discussed here, the Spring configuration is stored in just a single file, which is called `camel-context.xml`.

---

## Basic Spring configuration

[Example 5 on page 23](#) 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
       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 currently supported by Mediation 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 FUSE Mediation Router component and the `properties` element is used to set Bean properties on the component class.

[Example 6 on page 24](#) 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&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 [List of Components](#) in the *Component Reference*.

## 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/) [http://maven.apache.org/] or [Apache Ant](http://ant.apache.org/) [http://ant.apache.org/] to build your application.
2. Add the directory containing `META-INF/spring/*.xml` to your CLASSPATH.

---

### Running the application

Assuming that you have coded a `main()` method, as described in [Defining a Spring Main Method on page 22](#), 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/) [http://www.eclipse.org/] or [IntelliJ](http://www.jetbrains.com/idea/) [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.