Artix ESB

Java Router, Getting Started

Version 5.1
December 2007

Making Software Work Together™

Java Router, Getting Started

IONA Technologies

Version 5.1

Published 19 Dec 2007

Trademark and Disclaimer Notice

Copyright Notice

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

Table of Contents

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.

Introducing Java Router

Summary

This chapter describes the architecture of the Java Router and introduces some basic concepts that are important for understanding how the router works.

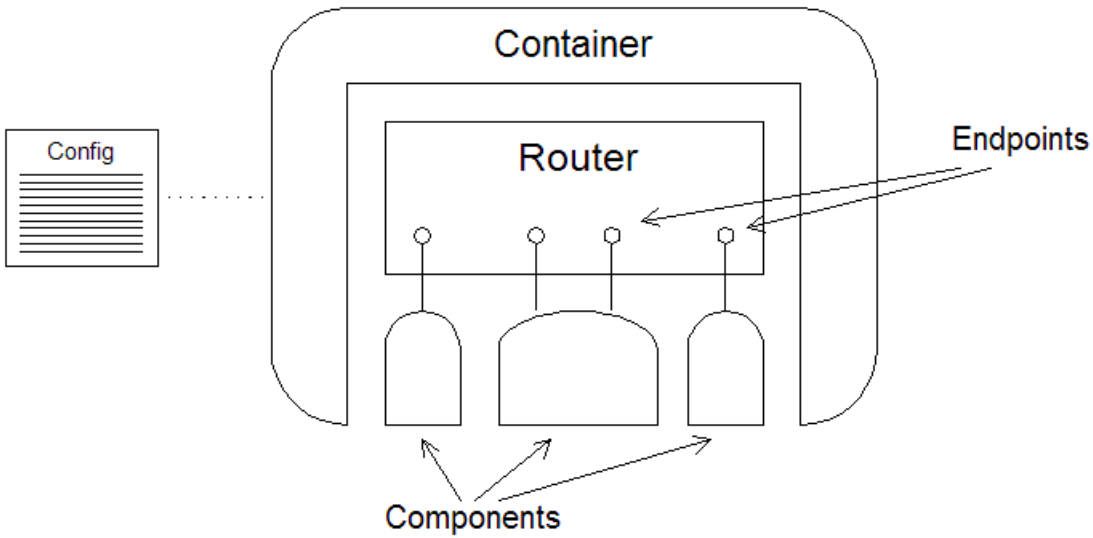
Table of Contents

Architecture

Overview

Figure 1, “Architecture of the Java Router” gives a general overview of the Java Router architecture. This architecture is designed with the basic requirement in mind that the router should be deployable in a wide variety of different container types.

Figure 1. Architecture of the Java Router



Router

The router service is represented by a *Camel context* object, which encapsulates routing rules (in the form of `RouteBuilder` objects) and components (which enable the router to bind to various network protocols and other resources). The router application itself consists either of Java code or XML configuration, or possibly a combination of the two.

Endpoints

In general, an endpoint is a specific source or a sink of messages, identified by a URI. In practice, this means that an endpoint maps either to a network location or to some other resource that can produce or absorb a stream of messages. Within a routing rule, endpoints are used in two distinct ways: the *source endpoint* appears at the start of a rule (for example, in a `from()`

command) and acts as a source of request messages and a sink for reply messages (if any); the *target endpoint* appears at the end of a rule (for example, in a `to()` command) and acts as a sink for request messages and a source of reply messages.

Components

A component is a plug-in that integrates the router core with a particular network protocol or external resource. From the perspective of a router developer, a component appears to be a factory for creating a specific type of endpoint. For example, there is a file component that can be used to create endpoints that read/write messages to and from particular directories or files. There is a CXF component that enables you to create endpoints that communicate with Web services (and related protocols).

Typically, before you can use a particular component, you need to configure it and add it to the Camel context. Some components, however, are embedded in the router core and do not need to be configured. The embedded components are as follows:

- Bean.
- Direct.
- File.
- JMX.
- Log.
- Mock.
- SEDA.
- Timer.
- VM.

For more details about the available components see the *Deployment Guide* and the list of Camel components [http://activemq.apache.org/camel/components.html].

RouteBuilders

The `RouteBuilder` classes encapsulate the routing rules. A router developer defines custom classes that inherit from `RouteBuilder` and adds instances of these classes to the `CamelContext`.

Deployment options

The router architecture is designed to facilitate deploying the router into different kinds of container. The following deployment options are currently supported:

- *Spring container deployment*—the router application is deployed into a Spring container and you can use the Spring configuration file to configure components and define routes.
- *Standalone deployment*—you must write a `main()` method in the application code, which is responsible for creating and registering `RouteBuilder` objects as well as configuring and registering components.

For more details about the deployment options, see the *Deployment Guide*.

Camel context

A `CamelContext` represents a single Camel routing rulebase. It defines the context used to configure routes and details which policies should be used during message exchanges between endpoints.

How to Develop a Router Application

Outline of the development steps

The following steps give a broad outline of what is involved in developing a router application:

1. *Choose a deployment option*—the router architecture is designed to support multiple deployment options. Currently, the following deployment options are supported:

   - Spring container deployment.
   - Standalone deployment.

2. *Define routing rules in Java DSL or in XML*—depending on the deployment option, you define the routing rules either in Java DSL or in XML.
3. *Configure components*—if you need to use components not already embedded in the router core, you must configure the components using either Java code or (in the case of a Spring container) XML.
4. *Deploy the router*—to deploy the router, follow the instructions for the particular container or deployment option that you have chosen. See the *Deployment Guide* for details.

Java Router Tutorial

Summary

This tutorial describes a Java Router demonstration in some detail, explaining the code for each part of the application and describing how to build and run the demonstration.

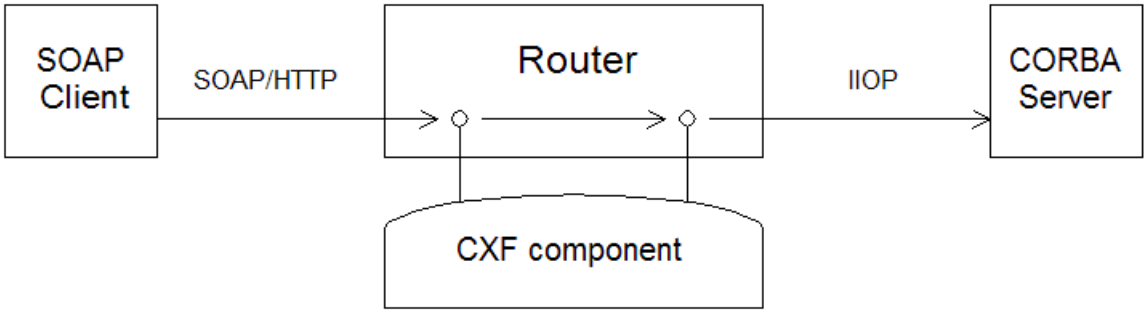
Table of Contents

Tutorial Overview

Overview

Figure 2, “Overview of the Java Router Tutorial” gives an overview of the *multibinding* router demonstration, which forms the basis for the tutorial in this chapter. Essentially, this demonstration shows how the router can switch messages between different binding/transport combinations. For example, in this demonstration, the incoming requests are received from the client in SOAP/HTTP format. The request messages are then routed to a CORBA server in IIOP (Internet Inter-ORB Protocol) format.

Figure 2. Overview of the Java Router Tutorial



To implement this route, the router requires just a single component: the CXF component, which is capable of supporting SOAP/HTTP endpoints as well as CORBA endpoints. The route itself is just a straightforward link between the source endpoint (SOAP/HTTP) and the target endpoint (CORBA). No additional processing is performed in the router.

Location of demonstration code

The code for the `multibinding` demonstration can be found in the following location:

```
ArtixInstallDir/java/samples/router/multibinding
```

Tutorial stages

The tutorial consists of the following stages:

- Tutorial: SOAP Client.
- Tutorial: CORBA Server.
- Tutorial: Router.

- Tutorial: Building and Running the Demonstration

Tutorial: SOAP Client

Overview

The SOAP client connects directly to the router, which exposes a SOAP/HTTP endpoint in order to listen for request messages from the client. The client is an entirely conventional JAX-WS client, implemented using the Artix Java runtime. The WSDL contract used by the client defines a `Greeter` interface (port type), a `GreeterService` service, and a `GreeterPort` port (for SOAP/HTTP). These basic features of the WSDL contract and the client application code are all described in this section.

Greeter port type

The `Greeter` port type in the WSDL contract, `samples/router/multibinding/greeter.wsdl`, defines the operations the SOAP client can call. The following operations are supported:

- `sayHi`—returns a salutation.
- `greetMe`—takes a string argument (the user's name) and returns a personalized greeting.
- `pingMe`—has no arguments or return value, but is capable of raising a fault exception.
- `greetMeOneWay`—has one string argument, but no return value (a oneway operation).

GreeterService service

Example 1, "GreeterService Service" shows the WSDL fragment for the SOAP service, `GreeterService`, that the client connects to.

Example 1. GreeterService Service

```
<wsdl:definitions name="Greeter"
targetNamespace="http://cxf.iona.com/demo/greeter"
    ...
    <wsdl:service name="GreeterService">
        <wsdl:port binding="tns:GreeterSOAPBinding"
name="GreeterPort">
            <soap:address
location="http://localhost:9090/GreeterContext/GreeterPort"/>

        </wsdl:port>
```

```
    </wsdl:service>
</wsdl:definitions>
```

From this WSDL fragment, we can read off the qualified names (QName) of the WSDL service and port. The value of the `targetNamespace` attribute in the `definitions` element determines the namespace of the service and port. Hence, the service and port QNames are as follows:

- Service QName is `{http://cxf.iona.com/demo/greeter}GreeterService`
- Port QName is `{http://cxf.iona.com/demo/greeter}GreeterPort`

Note

The value of the `location` attribute in the `soap:address` element is ignored, because the client overrides this address in the application code. See Example 2, “SOAP Client main() Method”.

Router address

Instead of connecting to the address given in the `wsdl:port` element (see Example 1, “GreeterService Service”), the client connects to the router process, which acts as a proxy for the CORBA service. The address that the client actually connects to is the following:

```
http://localhost:9000/router
```

Client main() method

The SOAP client is a standard JAX-WS client, implemented using the Artix Java runtime. The `main()` method creates a `Greeter` proxy in order to access the `GreeterService` service (as defined by the WSDL fragment, Example 1, “GreeterService Service”).

Example 2. SOAP Client main() Method

```
package demo.router.soap;

import javax.xml.ws.Service;
import static javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_BINDING;

import demo.router.Greeter;
```

```
import static demo.router.Constants.*;

public final class Client {
    ...
    public static void main(String args[]) throws Exception
{

        // create a service and get the router port
        Service service = Service.create(SERVICE_NAME); ❶
        service.addPort(PORT_NAME, SOAP11HTTP_BINDING,
ROUTER_TRANSPORT); ❷
        Greeter greeter = service.getPort(PORT_NAME,
Greeter.class);

        // make a invocation to the service
        String reply = greeter.greetMe("CORBA service"); ❸
        System.out.println(reply);
    }
}
```

The preceding code can be explained as follows:

❶ The `SERVICE_NAME` constant is defined to be `{http://cxf.iona.com/demo/greeter}GreeterService`.

❷ The `PORT_NAME` constant is defined to be `{http://cxf.iona.com/demo/greeter}GreeterPort`. The `ROUTER_TRANSPORT` constant is defined to be `http://localhost:9000/router` and this value overrides the address that appears in the `soap:address` element of the WSDL contract.

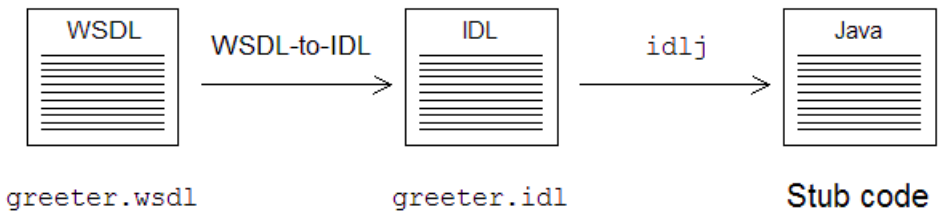
❸ Invoke the remote `greetMe` operation, using the `greeter` proxy object.

Tutorial: CORBA Server

Overview

In this scenario, the CORBA server responds to IIOP request messages from the router. The CORBA server is implemented using the Java version of Orbix. Because the server is a fully-fledged CORBA application, it is necessary to describe the Greeter interface in terms of OMG IDL (Interface Definition Language) in order to generate the requisite CORBA stub code. Figure 3, "Generating CORBA Stub Code" gives a schematic overview of how the stub code is generated in this demonstration.

Figure 3. Generating CORBA Stub Code



The starting point is a WSDL contract, `greeter.wsdl`, which is the contract used by the SOAP client (see Tutorial: SOAP Client). The Apache Ant build file calls the Yoko WSDL-to-IDL tool, `org.apache.yoko.tools.WSDLToIDL`, in order to convert this WSDL contract into an IDL file. The Java `idlj` tool is then called in order to generate stub code for the CORBA server.

CORBA server main() function

Example 3, "CORBA Server main() Function" shows an extract from the CORBA server's `main()` function. The server instantiates an object of `GreeterImpl` type, which implements the `Greeter` interface and makes it accessible through the URL, `corbaloc::localhost:40000/greeter`.

Example 3. CORBA Server main() Function

```
package demo.router.corba;

import java.util.Properties;

import com.iona.corba.IT_CORBA.WELL_KNOWN_ADDRESSING_POLICY_ID;
import com.iona.corba.IT_PlainTextKey.Forwarder;
```

```
import
com.iona.corba.IT_PortableServer.PERSISTENCE_MODE_POLICY_ID;
import
com.iona.corba.IT_PortableServer.PersistenceModePolicyValue;
import
com.iona.corba.IT_PortableServer.PersistenceModePolicyValueHelper;

import org.omg.CORBA.Any;
import org.omg.CORBA.ORB;
import org.omg.CORBA.Policy;
import org.omg.CORBA.UserException;

import org.omg.PortableServer.IdAssignmentPolicyValue;
import org.omg.PortableServer.LifespanPolicyValue;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;
import org.omg.PortableServer.POAManager;

public class Server {
    org.omg.CORBA.ORB orb;

   private static POA createPOA(String name, org.omg.CORBA.ORB
 orb,
                                 POA rootPOA, POAManager
rootPOAManager, String wakey) { ❶
        // Create POA with PERSISTENT life span,
DIRECT_PERSISTENCE persistence mode,
        // and a well-known addressing policy.
        // ... (not shown)
    }

    int runCorbaServer() throws UserException {
        // Resolve Root POA
        POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
 ❷

        // Get a reference to the POA manager
        POAManager manager = rootPOA.the_POAManager(); ❸
        POA poa = createPOA("hello_world", orb, rootPOA,
manager, "greeter");

        GreeterImpl hwImpl = new GreeterImpl(poa); ❹

        byte[] oid = "GreeterServer".getBytes();
        poa.activate_object_with_id(oid, hwImpl); ❺

        org.omg.CORBA.Object obj =
```

```
poa.create_reference_with_id(oid, GreeterHelper.id()); ❻
        Greeter hello = GreeterHelper.narrow(obj);

        try  {
            Forwarder forwarder =
(Forwarder)orb.resolve_initial_references("IT_PlainTextKeyForwarder");

            forwarder.add_plain_text_key("greeter", hello);
❼
        } catch (Exception ex) {
            throw new RuntimeException("Could not register
the corbaloc address");
        }

        // Run implementation
        manager.activate(); ❽
        System.out.println("Server ready...");
        orb.run();

        return 0;
    }

    public void run() {
        java.util.Properties props = new Properties();
        props.putAll(System.getProperties());
        props.put("org.omg.CORBA.ORBClass",
"com.iona.corba.art.artimpl.ORBImpl"); ❾
        props.put("org.omg.CORBA.ORBSingletonClass",
                  "com.iona.corba.art.artimpl.ORBSingleton");


        String[] arguments = new String[10]; ❿
        arguments[0] =  "-ORBdomain_name";
        arguments[1] = "artix";
        arguments[2] = "-ORBgreeter:iiop:host";
        arguments[3] = "localhost";
        arguments[4] = "-ORBgreeter:iiop:port";
        arguments[5] = "40000";

        try {
            orb = ORB.init(arguments, props);
            runCorbaServer();
        } catch (Exception ex) {
            ex.printStackTrace();
        }

        // ...
    }
```

```
    public void shutdownServer() {
        if (orb != null) {
            try {
                orb.shutdown(false);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        try {
            Server s = new Server();
            s.run();
        } catch (Exception ex) {
            ex.printStackTrace();
            System.exit(-1);
        } finally {
            System.out.println("done!");
        }
    }
}
```

Where the preceding C++ code can be explained as follows:

❶ The `createPOA()` method creates an instance of a POA (Portable Object Adapter) which represents a collection of CORBA objects that share the same set of policies. In particular, all of the CORBA objects in a POA share the same endpoint address. This method creates a POA instance with the following policy settings:

- Life-span policy is `PERSISTENT`—which means that object references remain valid even after you stop and restart the server.
- ID assignment policy is `USER_ID`—which means that the developer specifies object identifiers explicitly, instead of relying on the ORB to auto-generate one.
- Persistence mode policy is `DIRECT_PERSISTENT`—which means that clients establish connections *directly* to the server, instead of going through the Orbix locator.
- Well-known addressing policy is `greeter`—which enables you to configure the CORBA endpoint's address using configuration variables of the form `greeter:iiop:host` and `greeter:iiop:port`.

❷ Get a reference to the root POA object, `rootPOA`, using the ORB initialization service. The root POA is a factory for POA objects.

❸ Get a reference to the POA manager, which is used to start and stop the CORBA service.

Note

Every POA instance has an associated POA manager. Because of the hierarchical relationship between POA instances, the root POA's POA manager can be used to start or stop *all* of the endpoints (that is, POA instances) simultaneously.

❹ Create the object (of `GreeterImpl` type), that implements the `Greeter` interface.

❺ Associate the `GreeterImpl` object with the CORBA endpoint (POA instance). In CORBA terminology, this is called *activating* the object. When you activate the object, you also assign its *object ID* (which gets embedded into the object reference).

❻ The following two lines create an *object reference* for the Greeter CORBA object. The object reference, `hello`, encapsulates the information that is needed to remotely access the Greeter CORBA object (for example, clients would obtain the object reference in order to access the CORBA object).

❼ The code in this `try` block initializes a plain text key, which is used as an alias for the object reference. The purpose of this code is to enable you to use a plain, human-readable URL to access the CORBA service. In this example, the URL is `corbaloc::localhost:40000/greeter`. By setting the forwarder as shown, you can specify the `corbaloc` URL using `/greeter` in place of an unreadable object key.

❽ Start the CORBA service by calling `activate()` on the root POA manager. This has the effect of spawning a sub-thread that starts to listen for incoming CORBA request messages.

❾ Set the system properties for passing to the `ORB.init()` method. The effect of setting the `org.omg.CORBA.ORBClass` property and the `org.omg.CORBA.ORBSingletonClass` property as shown is that the Java interpreter uses the Orbix implementation of `org.omg.CORBA.ORB` (if you did not set these properties as shown,

the ORB implementation would default to Sun's implementation, which is embedded in the Java runtime).

Note

The `org.omg.CORBA.ORBSingletonClass` property setting is actually ineffective, because the ORB singleton object is already instantiated at this point. This is not important for this demonstration, because the the ORB singleton object is not needed here.

⑩ The ORB initialization parameters defined here are used to configure the Orbix ORB. In particular, the settings for `greeter:iiop:host` and `greeter:iiop:port` specify the physical address of the CORBA endpoint (POA instance). Recall that the well-known addressing policy (set in the `createPOA()` method) assocates the prefix, `greeter:`, with the CORBA endpoint.

Tutorial: Router

Overview

The router in the `multibinding` demonstration implements a simple routing rule: a SOAP/HTTP source endpoint (which listens for operation invocations from a SOAP client) is linked to a CORBA target endpoint. The purpose of this demonstration is to show that the router can route messages between different protocol bindings, transforming and redirecting the messages as it does so.

CXF component

The CXF component provides the router with access to the Artix Java runtime. In particular, the CXF component provides access to all of the bindings and transports supported by the Java runtime, including SOAP/HTTP, SOAP/JMS, CORBA, and so on. Using the CXF component, you can define endpoint URIs for any of these bindings and transports (where the URI prefix is `cxf://`).

Note

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

Source endpoint

The source endpoint for the routing rule is given by the following CXF endpoint URI:

```
cxf://http://localhost:9000/router
    ?serviceClass=demo.router.Greeter&dataFormat=POJO
```

Target endpoint

The target endpoint for the routing rule is given by the following CXF endpoint URI:

```
cxf://corbaloc::localhost:40000/greeter
    ?serviceClass=demo.router.Greeter&dataFormat=POJO

&portName={http://cxf.iona.com/demo/greeter}GreeterCORBAPort

&serviceName={http://cxf.iona.com/demo/greeter}GreeterCORBAService
```

```
    &wsdlURL=greeter-corba.wsdl
```

Router code

Example 4, “Multibinding Router Code” shows the contents of the `Router.java` file, which implements a standalone router. The router defines a single rule to route messages from SOAP/HTTP source endpoint to a CORBA target endpoint.

Example 4. Multibinding Router Code

```
package demo.router;

import org.apache.camel.CamelContext;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.impl.DefaultCamelContext;

public class Router {

    protected static final String URI_QRY =
        "?serviceClass=demo.router.Greeter&dataFormat=POJO";

    protected RouteBuilder createRouteBuilder() {
        return new RouteBuilder() { ❶
            public void configure() {
                from(getFromURI()).to(getToURI()); ❷
            }
        };
    }

    protected CamelContext createCamelContext() throws
Exception {
        return new DefaultCamelContext();
    }

    protected String getFromURI() { ❸
       return "cxf://" + Constants.ROUTER_TRANSPORT + URI_QRY;

    }

    protected String getToURI() { ❹
        return "cxf://" + Constants.SVC_TRANSPORT + URI_QRY
            +
"&portName={http://cxf.iona.com/demo/greeter}GreeterCORBAPort"

            +
"&serviceName={http://cxf.iona.com/demo/greeter}GreeterCORBAService"
```

```
            + "&wsdlURL=greeter-corba.wsdl";
    }

    public void run() throws Exception { ❺
        CamelContext context = createCamelContext();
        context.addRoutes(createRouteBuilder());
        context.start();
    }

    public static void main(String [] args) throws Exception
 {
        Router router = new Router();
        router.run();
        System.out.println("Router ready...");
    }
}
```

The preceding code can be explained as follows:

❶ Create a `org.apache.camel.builder.RouteBuilder` object to define the router rules. This line of code uses the Java syntax for creating and instantiating a class on the fly.

❷ This router has just one routing rule: connect a SOAP/HTTP endpoint (from which it receives client requests) to a CORBA endpoint.

❸ The `getFromURI()` method constructs the URI for the source endpoint, which is a SOAP/HTTP endpoint. See Source endpoint for details.

❹ The `getToURI()` method constructs the URI for the target endpoint, which is a CORBA endpoint. See Target endpoint for details.

❺ The code in the `Router.run()` method follows the usual outline for a standalone deployment of the router: it creates a Camel context, adds a `RouteBuilder` object, and then starts the router service.

Tutorial: Building and Running the Demonstration

Overview

In this stage of the tutorial, you will use the Apache Ant [http://ant.apache.org/] build tool to build and run the `multibinding` demonstration.

Steps to run and build the multibinding demonstration

Perform the following steps to build and run the `multibinding` demonstration:

1. You must build the demonstration using the Java runtime. If you have not already done so, you need to configure your environment to use the Java runtime. For each command window that you open, you should enter the following:

   - **Windows**

     ```
     > ArtixInstallDir\java\bin\artix_java_env.bat
     ```

   - **UNIX** (for a bourne-compatible shell)

     ```
     % . ArtixInstallDir/java/bin/artix_java_env
     ```

2. To build the demonstration, open a command window, change directory to `samples/router/multibinding` and enter the following command:

   ```
   ant build
   ```

3. To run the CORBA Greeter service, enter the following command:

   ```
   ant server
   ```

4. To run the router, enter the following command:

   ```
   ant router
   ```

5. To run the SOAP client, enter the following command:

   ```
   ant client
   ```

Defining Routes in Java DSL

Summary

You can define routing rules in Java, using a domain specific language (DSL). The routing rules represent the core of a router application and Java DSL is currently the most flexible way to define them.

Table of Contents

Implementing a RouteBuilder Class

Overview

In Java Router, you define routes by implementing a `RouteBuilder` class. You must override a single method, `RouteBuilder.configure()`, and in this method define the routing rules you want to associate with the `RouteBuilder`. The rules themselves are defined using a *Domain Specific Language* (DSL), which is implemented as a Java API.

You can define as many `RouteBuilder` classes as you like in a router application. Ultimately, each `RouteBuilder` class must get instantiated once and registered with the `CamelContext` object. Normally, however, the lifecycle of the `RouteBuilder` objects is managed automatically by the container in which you deploy the router. The core task for a router developer is simply to implement one or more `RouteBuilder` classes.

RouteBuilder class

The `org.apache.camel.builder.RouteBuilder` class is the base class for implementing your own route builder types. It defines an abstract method, `configure()`, that you *must* override in your derived implementation class. In addition, `RouteBuilder` also defines methods that are used to initiate the routing rules (for example, `from()`, `intercept()`, and `exception()`).

Implementing a RouteBuilder

Example 5, “Implementation of a RouteBuilder Class” shows an example of a simpler `RouteBuilder` implementation. You need only define a single method, `configure()`, which contains a list of routing rules (one Java statement for each rule).

Example 5. Implementation of a RouteBuilder Class

```
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {

    public void configure() {
        // Define routing rules here:

from("file:src/data?noop=true").to("file:target/messages");
```

```
        // More rules can be included, in you like.
        // ...
    }
}
```

Where the rule of the form `from(URL1).to(URL2)` instructs the router to read messages from the file system located in directory, `src/data`, and send them to files located in the directory, `target/messages`. The option, `?noop=true`, specifies that the source messages are *not* to be deleted from the `src/data` directory.

Basic Java DSL Syntax

What is a DSL?

A Domain Specific Language (DSL) is essentially a mini-language designed for a special purpose. The DSL is not required to be logically complete; it need only have enough expressive power to describe problems adequately in the chosen domain.

Typically, a DSL does *not* require a dedicated parser, interpreter, or compiler. You can piggyback a DSL on top of an existing object-oriented host language by observing that it is possible to map an API in a host language to a specialized language syntax: that is, a sequence of commands in the DSL maps to a chain of method invocations in the host language. For example, a sequence of commands in some hypothetical DSL that might look like this:

```
command01;
command02;
command03;
```

Can be mapped to a chain of Java invocations, like this:

```
command01().command02().command03()
```

You could even define blocks, for example:

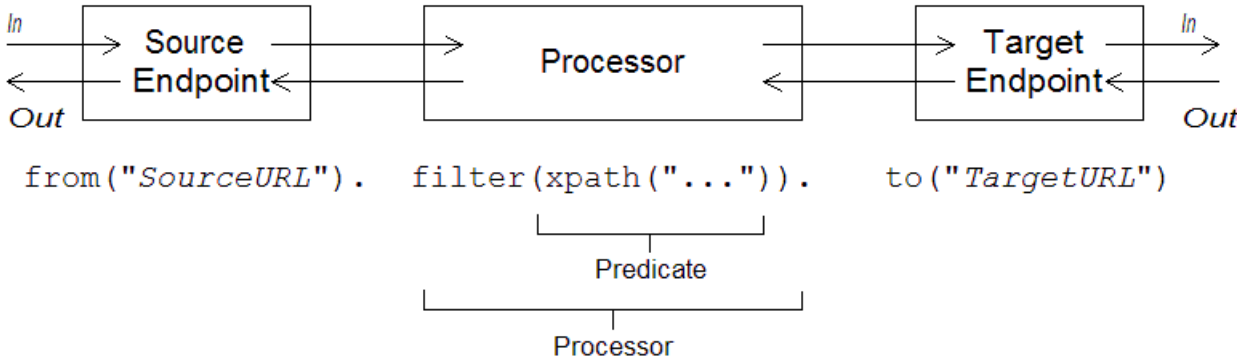
```
command01().startBlock().command02().command03().endBlock()
```

The syntax of the DSL is implicitly defined by the type system of the specialized API. For example, the return type of a method determines which methods can legally be invoked next (equivalent to the next command in the DSL).

Router rule syntax

The Java Router defines a *router DSL* for defining routing rules. You can use this DSL to define rules in the body of a `RouteBuilder.configure()` implementation. Figure 4, “Local Routing Rules” shows an overview of the basic syntax for defining local routing rules.

Figure 4. Local Routing Rules



A local rule always starts with a `from("EndpointURL")` method, which specifies the source of messages for the routing rule. You can then add an arbitrarily long chain of processors to the rule (for example, `filter()`), finishing off the rule with a `to("EndpointURL")` method, which specifies the target for the messages that pass through the rule. It is not always necessary to end a rule with `to()`, however. There are alternative ways of specifying the message target in a rule.

Note

It is also possible to define a global routing rule, by starting the rule with a special processor type (such as `intercept()`, `exception()`, `errorHandler()`, and so on). This kind of rule lies outside the scope of the *Getting Started* guide.

Sources and targets

A local rule always starts by defining a source endpoint, using `from("EndpointURL")`, and typically (but not always) ends by defining a target endpoint, using `to("EndpointURL")`. The endpoint URLs, *`EndpointURL`*, can use any of the components configured at deploy time. For example, you could use a file endpoint, `file:MyMessageDirectory`, a CXF endpoint, `cxf:MyServiceName`, or an ActiveMQ endpoint,

`activemq:queue:MyQName`. For a complete list of component types, see http://activemq.apache.org/camel/components.html.

Processors

A *processor* is a method that can access and modify the stream of messages passing through a rule. If a message is part of a remote procedure call (*InOut* call), the processor can potentially act on the messages flowing in *both* directions: on the request messages, flowing from source to target, and on the reply messages, flowing from target back to source (see Message exchanges). Processors can take *expression* or *predicate* arguments, that modify their behavior. For example, the rule shown in Figure 4, "Local Routing Rules" includes a `filter()` processor that takes an `xpath()` predicate as its argument.

Expressions and predicates

Expressions (evaluating to strings or other data types) and predicates (evaluating to true or false) occur frequently as arguments to the built-in processor types. You do not have to worry much about which type to pass to an expression argument, because they are usually automatically converted to the type you need. For example, you can usually just pass a string into an expression argument. Predicate expressions are useful for defining conditional behaviour in a route. For example, the following filter rule propagates *In* messages, only if the `foo` header is equal to the value `bar`:

```
from("seda:a").filter(header("foo").isEqualTo("bar")).to("seda:b");
```

Where the filter is qualified by the predicate, `header("foo").isEqualTo("bar")`. To construct more sophisticated predicates and expressions, based on the message content, you can use one of the expression and predicate languages (see Languages for Expressions and Predicates).

Message exchanges

When a router rule is activated, it can process messages passing in either direction: that is, from source to target or from target back to source. For example, if a router rule is mediating a remote procedure call (RPC), the rule would process requests, replies, and faults. How do you manage message correlation in this case? One of the most effective and straightforward ways is to use a *message exchange* object as the basis for processing messages. Java Router uses message exchange objects (of `org.apache.camel.Exchange` type) in its API for processing router rules.

The basic idea of the message exchange is that, instead of accessing requests, replies, and faults separately, you encapsulate the correlated messages inside

a single object (an `Exchange` object). Message correlation now becomes trivial from the perspective of a processor, because correlated messages are encapsulated in a single `Exchange` object and processors gain access to messages through the `Exchange` object.

Using an `Exchange` object makes it easy to generalize message processing to different kinds of *message exchange pattern*. For example, an asynchronous protocol might define a message exchange pattern that consists of a single message that flows from the source to the target (an *In* message). An RPC protocol, on the other hand, might define a message exchange pattern that consists of a request message correlated with either a reply or fault message. Currently, Java Router supports the following message exchange patterns:

- `InOnly`
- `RobustInOnly`
- `InOut`
- `InOptionalOut`
- `OutOnly`
- `RobustOutOnly`
- `OutIn`
- `OutOptionalIn`

Where these message exchange patterns are represented by constants in the enumeration type, `org.apache.camel.ExchangePattern`.

Processors

Overview

To enable the router to something more interesting than simply connecting a source endpoint to a target endpoint, you can add *processors* to your route. A processor is a command you can insert into a routing rule in order to perform arbitrary processing of the messages that flow through the rule. Java Router provides a wide variety of different processors, as follows:

- Filter.
- Choice.
- Pipeline
- Recipient list.
- Splitter.
- Aggregator.
- Resequencer.
- Throttler.
- Delayer.
- Load balancer
- Custom processor.

Filter

The `filter()` processor can be used to prevent uninteresting messages from reaching the target endpoint. It takes a single predicate argument: if the predicate is true, the message exchange is allowed through to the target; if the predicate is false, the message exchange is blocked. For example, the following filter blocks a message exchange, unless the incoming message contains a header, `foo`, with value equal to `bar`:

```
from("SourceURL").filter(header("foo").isEqualTo("bar")).to("TargetURL");
```

Choice

The `choice()` processor is a conditional statement that is used to route incoming messages to alternative targets. The alternative targets are each

preceded by a `when()` method, which takes a predicate argument. If the predicate is true, the following target is selected, otherwise processing proceeds to the next `when()` method in the rule. For example, the following `choice()` processor directs incoming messages to either *`Target1`*, *`Target2`*, or *`Target3`*, depending on the values of *`Predicate1`* and *`Predicate2`*:

```
from("SourceURL").choice().when(Predicate1).to("Target1")
                          .when(Predicate2).to("Target2")
                          .otherwise().to("Target3");
```

Pipeline

The `pipeline()` processor is used to link together a chain of targets, where the output of one target is fed into the input of the next target in the pipeline (analogous to the UNIX `pipe` command). The `pipeline()` method takes an arbitrary number of endpoint arguments, which specify the sequence of endpoints in the pipeline. For example, to pass messages from *`SourceURL`* to *`Target1`* to *`Target2`* to *`Target3`* in a pipeline, you could use the following rule:

```
from("SourceURL").pipeline("Target1","Target2","Target3");
```

Recipient list

If you want the messages from a source endpoint, *`SourceURL`*, to be sent to more than one target, there are two alternative approaches you can use. One approach is to invoke the `to()` method with multiple target endpoints (static recipient list), for example:

```
from("SourceURL").to("Target1","Target2","Target3");
```

The alternative approach is to invoke the `recipientList()` processor, which takes a list of recipients as its argument (dynamic recipient list). The advantage of the `recipientList()` processor is that the list of recipients can be calculated at runtime. For example, the following rule generates a recipient list by reading the contents of the `recipientListHeader` from the incoming message:

```
from("SourceURL").recipientList(header("recipientListHeader").tokenize(","));
```

Splitter

The `splitter()` processor is used to split a message into parts, which are then processed as separate messages. The `splitter()` method takes a list argument, where each item in the list represents a message part that is to be re-sent as a separate message. For example, the following rule splits the body of an incoming message into separate lines and then sends each line to the target in a separate message:

```
from("SourceURL").splitter(bodyAs(String.class).tokenize("\n")).to("TargetURL");
```

Aggregator

The `aggregator()` processor is used to aggregate related incoming messages into a single message. In order to distinguish which messages are eligible to be aggregated together, you need to define a *correlation key* for the aggregator. The correlation key is normally derived from a field in the message (for example, a header field). Messages that have the same correlation key value are eligible to be aggregated together. You can also optionally specify an aggregation algorithm to the `aggregator()` processor (the default algorithm is to pck the latest message with a given value of the correlation key and to discard the older messages with that correlation key value).

For example, if you are monitoring a data stream that reports stock prices in real time, you might only be interested in the *latest* price of each stock symbol. In this case, you could configure an aggregator to transmit only the latest price for a given stock and discard the older (out-of-date) price notifications. The following rule implements this functionality, where the correlation key is read from the `stockSymbol` header and the default aggregator algorithm is used:

```
from("SourceURL").aggregator(header("stockSymbol")).to("TargetURL");
```

Resequencer

A `resequencer()` processor is used to re-arrange the order in which incoming messages are transmitted. The `resequencer()` method takes a sequence number as its argument (where the sequence number is calculated from the contents of a field in the incoming message). Naturally, before you can start re-ordering messages, you need to wait until a certain number of messages have been received from the source. There are a couple of different ways to specify how long the `resequencer()` processor should wait before

attempting to re-order the accumulated messages and forward them to the target, as follows:

- *Batch resequencing*—(the default) wait until a specified number of messages have accumulated before starting to re-order and forward messages. This processing option is specified by invoking `resequencer().batch()`. For example, the following resequencing rule would re-order messages based on the `timeOfDay` header, waiting until at least 300 messages have accumulated or 4000 ms have elapsed since the last message received.

```
from("SourceURL").resequencer(header("timeOfDay").batch(new
 BatchResequencerConfig(300, 4000L)).to("TargetURL");
```

- *Stream resequencing*—transmit messages as soon as they ariive *unless* the resequencer detects a gap in the incoming message stream (missing sequence numbers), in which case the resequencer waits until the missing messages arrive and then forwards the messages in the correct order. To avoid the resequencer blocking forever, you can specify a timeout (default is 1000 ms), after which time the message sequence is transmitted with unresolved gaps. For example, the following resequencing rule detects gaps in the message stream by monitoring the value of the `sequenceNumber` header, where the maximum buffer size is limited to 5000 and the timeout is specified to be 4000 ms:

```
from("SourceURL").resequencer(header("sequenceNumber")).stream(new
 StreamResequencerConfig(5000, 4000L)).to("TargetURL");
```

Throttler

The `throttler()` processor is used to ensure that a target endpoint does not get overloaded. The throttler works by limiting the number of messages that can pass through per second. If the incoming messages exceed the specified rate, the throttler accumulates excess messages in a buffer and transmits them more slowly to the target endpoint. For example, to limit the rate of throughput to 100 messages per second, you can define the following rule:

```
from("SourceURL").throttler(100).to("TargetURL");
```

Delayer

The `delayer()` processor is used to hold up messages for a specified length of time. The delay can either be relative (wait a specified length of time after

receipt of the incoming message) or absolute (wait until a specific time). For example, to add a delay of 2 seconds before transmitting received messages, you can use the following rule:

```
from("SourceURL").delayer(2000).to("TargetURL");
```

To wait until the absolute time specified in the `processAfter` header, you can use the following rule:

```
from("SourceURL").delayer(header("processAfter").to("TargetURL");
```

The `delayer()` method is overloaded, such that an integer is interpreted as a relative delay and an expression (for example, a string) is interpreted as an absolute delay.

Load balancer

The `loadBalance()` processor is used to load balance message exchanges over a list of target endpoints. It is possible to customize the load balancing strategy. For example, to load balance incoming messages exchanges using a round robin algorithm (each endpoint in the target list is tried in sequence), you can use the following rule:

```
from("SourceURL").loadBalance().roundRobin().to("TargetURL_01",
 "TargetURL_02", "TargetURL_03");
```

Alternatively, you can customize the load balancing algorithm by implementing your own `LoadBalancer` class, as follows:

```
public class MyLoadBalancer implements
org.apache.camel.processor.loadbalancer.LoadBalancer {
    ...
};

from("SourceURL").loadBalance().setLoadBalancer(new
MyLoadBalancer())
    .to("TargetURL_01", "TargetURL_02", "TargetURL_03");
```

Custom processor

If none of the standard processors described here provide the functionality you need, you can always define your own custom processor. To create a custom processor, define a class that implements the `org.apache.camel.Processor` interface and override the `process()` method in this class. For example, the following custom processor, `MyProcessor`, removes the header named `foo` from incoming messages:

Example 6. Implementing a Custom Processor Class

```
public class MyProcessor implements org.apache.camel.Processor
 {
    public void process(org.apache.camel.Exchange exchange)
{
        inMessage = exchange.getIn();
        if (inMessage != null) {
            inMessage.removeHeader("foo");
        }
    }
};
```

To insert the custom processor into a router rule, invoke the `process()` method, which provides a generic mechanism for inserting processors into rules. For example, the following rule invokes the processor defined in Example 6, “Implementing a Custom Processor Class”:

```
org.apache.camel.Processor myProc = new MyProcessor();

from("SourceURL").process(myProc).to("TargetURL");
```

Languages for Expressions and Predicates

Overview

To provide you with greater flexibility when parsing and processing messages, Java Router supports language plug-ins for various scripting languages. For example, if an incoming message is formatted as XML, it is relatively easy to extract the contents of particular XML elements or attributes from the message using a language such as XPath. The Java Router implements script builder classes, which encapsulate the imported languages. Each languages is accessed through a static method that takes a script expression as its argument, processes the current message using that script, and then returns an expression or a predicate. In order to be usable as an expression or a predicate, the script builder classes implement the following interfaces:

```
org.apache.camel.Expression<E>
org.apache.camel.Predicate<E>
```

In addition to this, the `ScriptBuilder` class (which wraps scripting languages such as JavaScript, and so on) inherits from the following interface:

```
org.apache.camel.Processor
```

Which implies that the languages associated with the `ScriptBuilder` class can also be used as message processors (see Custom processor).

Simple

The simple language is a very limited expression language that is built into the router core. This language can be useful, if you need to eliminate dependancies on third-party libraries whilst testing. Otherwise, you should use one of the other languages. To use the simple language in your application code, include the following import statement in your Java source files:

```
import org.apache.camel.language.simple.SimpleLanguage.simple;
```

The simple language provides various elementary expressions that return different parts of a message exchange. For example, the expression, `simple("header.timeOfDay")`, would return the contents of a header called `timeOfDay` from the incoming message. You can also construct predicates by testing expressions for equality. For example, the predicate, `simple("header.timeOfDay = '14:30'")`, tests whether the `timeOfDay` header in the incoming message is equal to `14:30`. Table 1, “Properties for Simple Language” shows the list of elementary expressions supported by the simple language.

Table 1. Properties for Simple Language

Elementary Expression	Description
`body`	Access the body of the incoming message.
`out.body`	Access the body of the outgoing message.
`header.`*`HeaderName`*	Access the contents of the *`HeaderName`* header from the incoming message.
`out.header.`*`HeaderName`*	Access the contents of the *`HeaderName`* header from the outgoing message.
`property.`*`PropertyName`*	Access the *`PropertyName`* property on the exchange.

XPath

The `xpath()` static method parses message content using the XPath language (to learn about XPath, see the W3 Schools tutorial, http://www.w3schools.com/xpath/default.asp). To use the XPath language in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.xml.XPathBuilder.xpath;
```

You can pass an XPath expression to `xpath()` as a string argument. The XPath expression implicitly acts on the message content and returns a node set as its result. Depending on the context, the return value is interpreted either as a predicate (where an empty node set is interpreted as false) or an expression. For example, if you are processing an XML message with the following content:

```
<person user="paddington">
  <firstName>Paddington</firstName>
  <lastName>Bear</lastName>
  <city>London</city>
</person>
```

You could choose which target endpoint to route the message to, based on the content of the `city` element, using the following rule:

```
from("file:src/data?noop=true").
    choice().
        when(xpath("/person/city =
'London'")).to("file:target/messages/uk").
        otherwise().to("file:target/messages/others");
```

Where the return value of `xpath()` is treated as a predicate in this example.

XQuery

The `xquery()` static method parses message content using the XQuery language (to learn about XQuery, see the W3 Schools tutorial, http://www.w3schools.com/xquery/default.asp). XQuery is a superset of the XPath language; hence, any valid XPath expression is also a valid XQuery expression. To use the XQuery language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.saxon.XQueryBuilder.xquery;
```

You can pass an XQuery expression to `xquery()` in several different ways. For simple expressions, you can pass the XQuery expressions as a string, `java.lang.String`. For longer XQuery expressions, on the other hand, you might prefer to store the expression in a file, which you can then reference by passing a `java.io.File` argument or a `java.net.URL` argument to the overloaded `xquery()` method. The XQuery expression implicitly acts on the message content and returns a node set as its result. Depending on the context, the return value is interpreted either as a predicate (where an empty node set is interpreted as false) or an expression.

JoSQL

The `sql()` static method enables you to call on the JoSQL (SQL for Java objects) language to evaluate predicates and expressions in Java Router. JoSQL employs a SQL-like query syntax to perform selection and ordering operations on data from in-memory Java objects—JoSQL is *not* a database, however. In the JoSQL syntax, each Java object instance is treated like a table row and each object method is treated like a column name. Using this syntax, it is possible to construct powerful statements for extracting and compiling data from collections of Java objects. For details, see http://josql.sourceforge.net/.

To use the JoSQL language in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.sql.SqlBuilder.sql;
```

OGNL

The `ognl()` static method enables you to call on OGNL (Object Graph Navigation Language) expressions, which can then be used as predicates and expressions in a router rule. For details, see http://www.ognl.org/.

To use the OGNL language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.language.ognl.OgnlExpression.ognl;
```

EL

The `el()` static method enables you to call on the Unified Expression Language (EL) to construct predicates and expressions in a router rule. The EL was originally specified as part of the JSP 2.1 standard (JSR-245), but is now available as a standalone language. Java Router integrates with JUEL (http://juel.sourceforge.net/), which is an open source implementation of the EL language.

To use the EL language in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.language.juel.JuelExpression.el;
```

Groovy

The `groovy()` static method enables you to call on the Groovy scripting language to construct predicates and expressions in a route. To use the Groovy language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.camel.script.ScriptBuilder.*;
```

JavaScript

The `javaScript()` static method enables you to call on the JavaScript scripting language to construct predicates and expressions in a route. To use the JavaScript language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.camel.script.ScriptBuilder.*;
```

PHP

The `php()` static method enables you to call on the PHP scripting language to construct predicates and expressions in a route. To use the PHP language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.camel.script.ScriptBuilder.*;
```

Python

The `python()` static method enables you to call on the Python scripting language to construct predicates and expressions in a route. To use the Python language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.camel.script.ScriptBuilder.*;
```

Ruby

The `ruby()` static method enables you to call on the Ruby scripting language to construct predicates and expressions in a route. To use the Ruby language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.camel.script.ScriptBuilder.*;
```

Bean

You can also use Java beans to evaluate predicates and expressions. For example, to evaluate the predicate on a filter using the `isGoldCustomer()` method on the bean instance, `myBean`, you can use a rule like the following:

```
from("SourceURL")
        .filter().method("myBean", "isGoldCustomer")
        .to("TargetURL");
```

A discussion of bean integration in Java Router is beyond the scope of this *Getting Started* guide. For details, see http://activemq.apache.org/camel/bean-language.html.

Transforming Message Content

Overview

Java Router supports a variety of approaches to transforming message content. In addition to a simple native API for modifying message content, Java Router supports integration with serveral different third-party libraries and transformation standards. The following kinds of transformation are discussed in this section:

- Simple transformations.
- Marshalling and unmarshalling.
- Artix Data Services.

Simple transformations

The Java DSL has a built-in API that enables you to perform simple transformations on incoming and outgoing messages. For example, the rule shown in Example 7, “Simple Transformation of Incoming Messages” would append the text, `World!`, to the end of the incoming message body.

Example 7. Simple Transformation of Incoming Messages

```
from("SourceURL").setBody(body().append("
World!")).to("TargetURL");
```

Where the `setBody()` command replaces the content of the incoming message's body. You can use the following API classes to perform simple transformations of the message content in a router rule:

- `org.apache.camel.model.ProcessorType`
- `org.apache.camel.builder.Builder`
- `org.apache.camel.builder.ValueBuilder`

ProcessorType class

The `org.apache.camel.model.ProcessorType` class defines the DSL commands you can insert directly into a router rule—for example, the `setBody()` command in Example 7, “Simple Transformation of Incoming Messages”. Table 2, “Transformation Methods from the ProcessorType Class”

shows the `ProcessorType` methods that are relevant to transforming message content:

Table 2. Transformation Methods from the ProcessorType Class

Method	Description
`Type convertBodyTo(Class type)`	Converts the IN message body to the specified type.
`Type convertFaultBodyTo(Class type)`	Converts the FAULT message body to the specified type.
`Type convertOutBodyTo(Class type)`	Converts the OUT message body to the specified type.
`Type removeFaultHeader(String name)`	Adds a processor which removes the header on the FAULT message.
`Type removeHeader(String name)`	Adds a processor which removes the header on the IN message.
`Type removeOutHeader(String name)`	Adds a processor which removes the header on the OUT message.
`Type removeProperty(String name)`	Adds a processor which removes the exchange property.
`ExpressionClause<ProcessorType<Type>> setBody()`	Adds a processor which sets the body on the IN message.
`ExpressionClause<ProcessorType<Type>> setBody()`	Adds a processor which sets the body on the IN message.
`Type setFaultBody(Expression expression)`	Adds a processor which sets the body on the FAULT message.
`Type setFaultHeader(String name, Expression expression)`	Adds a processor which sets the header on the FAULT message.
`ExpressionClause<ProcessorType<Type>> setHeader(String name)`	Adds a processor which sets the header on the IN message.
`Type setHeader(String name, Expression expression)`	Adds a processor which sets the header on the IN message.
`ExpressionClause<ProcessorType<Type>> setOutBody()`	Adds a processor which sets the body on the OUT message.

Method	Description
`Type setOutBody(Expression expression)`	Adds a processor which sets the body on the OUT message.
`ExpressionClause<ProcessorType<Type>> setOutHeader(String name)`	Adds a processor which sets the header on the OUT message.
`Type setOutHeader(String name, Expression expression)`	Adds a processor which sets the header on the OUT message.
`ExpressionClause<ProcessorType<Type>> setProperty(String name)`	Adds a processor which sets the exchange property.
`Type setProperty(String name, Expression expression)`	Adds a processor which sets the exchange property.

Builder class

The `org.apache.camel.builder.Builder` class provides access to message content in contexts where expressions or predicates are expected. In other words, `Builder` methods are typically invoked in the *arguments* of DSL commands—for example, the `body()` command in Example 7, "Simple Transformation of Incoming Messages". Table 3, "Methods from the Builder Class" summarizes the static methods available in the `Builder` class.

Table 3. Methods from the Builder Class

Method	Description
`static <E extends Exchange> ValueBuilder<E> body()`	Returns a predicate and value builder for the inbound body on an exchange.
`static <E extends Exchange,T> ValueBuilder<E> bodyAs(Class<T> type)`	Returns a predicate and value builder for the inbound message body as a specific type.
`static <E extends Exchange> ValueBuilder<E> constant(Object value)`	Returns a constant expression.
`static <E extends Exchange> ValueBuilder<E> faultBody()`	Returns a predicate and value builder for the fault body on an exchange.

Method	Description
`static <E extends Exchange,T> ValueBuilder<E> faultBodyAs(Class<T> type)`	Returns a predicate and value builder for the fault message body as a specific type.
`static <E extends Exchange> ValueBuilder<E> header(String name)`	Returns a predicate and value builder for headers on an exchange.
`static <E extends Exchange> ValueBuilder<E> outBody()`	Returns a predicate and value builder for the outbound body on an exchange.
`static <E extends Exchange> ValueBuilder<E> outBody()`	Returns a predicate and value builder for the outbound message body as a specific type.
`static <E extends Exchange> ValueBuilder<E> systemProperty(String name)`	Returns an expression for the given system property.
`static <E extends Exchange> ValueBuilder<E> systemProperty(String name, String defaultValue)`	Returns an expression for the given system property.

ValueBuilder class

The `org.apache.camel.builder.ValueBuilder` class enables you to modify values returned by the `Builder` methods. In other words, the methods in `ValueBuilder` provide a simple way of modifying message content. Table 4, "Modifier Methods from the ValueBuilder Class" summarizes the methods available in the `ValueBuilder` class. That is, the table shows only the methods that are used to modify the value they are invoked on (for full details, see the *API Reference* documentation).

Table 4. Modifier Methods from the ValueBuilder Class

Method	Description
`ValueBuilder<E> append(Object value)`	Appends the string evaluation of this expression with the given value.
`ValueBuilder<E> convertTo(Class type)`	Converts the current value to the given type using the registered type converters.

Method	Description
`ValueBuilder<E> convertToString()`	Converts the current value a String using the registered type converters.
`ValueBuilder<E> regexReplaceAll(String regex, Expression<E> replacement)`	Replaces all occurrencies of the regular expression with the given replacement.
`ValueBuilder<E> regexReplaceAll(String regex, String replacement)`	Replaces all occurrencies of the regular expression with the given replacement.
`ValueBuilder<E> regexTokenize(String regex)`	Tokenizes the string conversion of this expression using the given regular expression.
`ValueBuilder<E> tokenize()`	
`ValueBuilder<E> tokenize(String token)`	Tokenizes the string conversion of this expression using the given token separator.

Marshalling and unmarshalling

You can convert between low-level and high-level message formats using the following commands:

- `marshal()`—convert a high-level data format to a low-level data format.
- `unmarshal()`—convert a low-level data format to a high-level data format.

Java Router supports marshalling and unmarshalling of the following data formats:

- *Java serialization*—enables you to convert a Java object to a blob of binary data. For this data format, unmarshalling converts a binary blob to a Java object and marshalling converts a Java object to a binary blob. For example, to read a serialized Java object from an endpoint, *`SourceURL`*, and convert it to a Java object, you could use the following rule:

```
from("SourceURL").unmarshal().serialization()
    .<FurtherProcessing>.to("TargetURL");
```

- *JAXB*—provides a mapping between XML schema types and Java types (see https://jaxb.dev.java.net/). For JAXB, unmarshalling converts an XML data type to a Java object and marshalling converts a Java object to an XML data type. Before you can use JAXB data formats, you must compile your XML schema using a JAXB compiler in order to generate the Java

classes that represent the XML data types in the schema. This is called *binding* the schema. After you have bound the schema, you can define a rule to unmarshal XML data to a Java object, using code like the following:

```
org.apache.camel.spi.DataFormat jaxb = new
org.apache.camel.model.dataformat.JaxbDataFormat("GeneratedPackageName");

from("SourceURL").unmarshal(jaxb)
    .<FurtherProcessing>.to("TargetURL");
```

Where `GeneratedPackagename` is the name of the Java package generated by the JAXB compiler, which contains the Java classes representing your XML schema.

- *XMLBeans*—provides an alternative mapping between XML schema types and Java types (see http://xmlbeans.apache.org/). For XMLBeans, unmarshalling converts an XML data type to a Java object and marshalling converts a Java object to an XML data type. For example, to unmarshal XML data to a Java object using XMLBeans, you can use code like the following:

```
from("SourceURL").unmarshal().xmlBeans()
    .<FurtherProcessing>.to("TargetURL");
```

- *XStream*—provides another mapping between XML types and Java types (see http://xstream.codehaus.org/). XStream is a serialization library (like Java serialization), enabling you to convert any Java object to XML. For XStream, unmarshalling converts an XML data type to a Java object and marshalling converts a Java object to an XML data type. For example, to unmarshal XML data to a Java object using XStream, you can use code like the following:

```
from("SourceURL").unmarshal().xstream()
    .<FurtherProcessing>.to("TargetURL");
```

- *Artix Data Services*—Java Router also integrates with Artix Data Services, enabling you to integrate stub code generated by Artix Data Services. See Artix Data Services for details.

Artix Data Services

Artix Data Services is a powerful tool for converting documents and messages between different data formats. In Artix Data Services, you can use a graphcial tool to define complex mapping rules (including processing of data content) and then generate stub code to implement the mapping rules (see IONA Artix

Data Services [http://www.iona.com/products/artix/data_services.htm?WT.mc_id=125795] home page and the Artix Data Services documentation [http://www.iona.com/support/docs/artix/data_services/3.6/index.xml] for more details). The `marshal()` and `unmarshal()` DSL commands are capable of consuming Artix Data Services stub code in order to perform transformations on message formats. For example, Example 8, "Using Artix Data Services to Marshal and Unmarshal" shows a rule that unmarshals XML documents into a canonical format (Java objects) and then marshals the canonical format into the tag/value pair format.

Example 8. Using Artix Data Services to Marshal and Unmarshal

```
from("SourceURL")
    . unmarshal().artixDS(DocumentElement.class,
ArtixDSContentType.Xml)
    . marshal().artixDS(ArtixDSContentType.TagValuePair)
        . to("TargetURL");
```

Note

Artix Data Services is licensed separately from Java Router.

Defining Routes in XML

Summary

You can define routing rules in XML. This approach is not as flexible as Java DSL, but has the advantage that it is easy to reconfigure the routing rules at runtime.

Table of Contents

Using the Router Schema in an XML File

Overview

The root element of the router schema is `camelContext`, which is defined in the XML namespace, `http://activemq.apache.org/camel/schema/spring`. Router configurations are typically embedded in other XML configuration files (for example, in a Spring configuration file). In general, whenever a router configuration is embedded in another configuration file, you need to specify the location of the router schema (so that the router configuration can be parsed). For example, Example 9, " Specifying the Router Schema Location" shows how to embed the router configuration, `camelContext`, in an arbitrary document, *`DocRootElement`*.

Example 9. Specifying the Router Schema Location

```
<DocRootElement ...
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://activemq.apache.org/camel/schema/spring
http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">


  <camelContext id="camel"
xmlns="http://activemq.apache.org/camel/schema/spring">
    <!-- Define your routing rules here -->
  </camelContext>
</DocRootElement>
```

Where the schema location is specified to be `http://activemq.apache.org/camel/schema/spring/camel-spring.xsd`, which gives the location of the schema on the Apache Web site. This location always contains the latest, most up-to-date version of the XML schema. If you prefer to tie your configuration to a specific version of the schema, change the schema file name to `camel-spring-`*`Version`*`.xsd`, where *`Version`* can be one of: `1.0.0`, `1.1.0`, or `1.2.0`. For example, the location of schema version 1.2.0 would be specified as `http://activemq.apache.org/camel/schema/spring/camel-spring-1.2.0.xsd`.

Example 10, "Router Schema in a Spring Configuration File" shows an example of embedding a router configuration, `camelContext`, in a Spring configuration file.

Example 10. Router Schema in a Spring Configuration File

```
<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

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


  <camelContext id="camel"
xmlns="http://activemq.apache.org/camel/schema/spring">
    <!-- Define your routing rules in here -->
  </camelContext>
  <!-- Other Spring configuration -->
  <!-- ... -->
</beans>
```

Defining a Basic Route in XML

Basic concepts

In order to understand how to build a route using XML, you need to understand some of the basic concepts of the routing language—for example, sources and targets, processors, expressions and predicates, and message exchanges. For definitions and explanations of these concepts see Basic Java DSL Syntax.

Example of a basic route

Example 11, "Basic Route in XML" shows an example of a basic route in XML, which connects a source endpoint, *SourceURL*, directly to a destination endpoint, *TargetURL*.

Example 11. Basic Route in XML

```
<camelContext id="CamelContextID"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

Where *CamelContextID* is an arbitrary, unique identifier for the Camel context. The route is defined by a `route` element and there can be multiple `route` elements under the `camelContext` element.

Processors

Overview

To enable the router to something more interesting than simply connecting a source endpoint to a target endpoint, you can add *processors* to your route. A processor is a command you can insert into a routing rule in order to perform arbitrary processing of the messages that flow through the rule. Java Router provides a wide variety of different processors, as follows:

- Filter.
- Choice.
- Recipient list.
- Splitter.
- Aggregator.
- Resequencer.
- Throttler.
- Delayer.

Filter

The `filter` processor can be used to prevent uninteresting messages from reaching the target endpoint. It takes a single predicate argument: if the predicate is true, the message exchange is allowed through to the target; if the predicate is false, the message exchange is blocked. For example, the following filter blocks a message exchange, unless the incoming message contains a header, `foo`, with value equal to `bar`:

```
<camelContext id="filterRoute"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <filter>
      <simple>header.foo = 'bar'</simple>
      <to uri="TargetURL"/>
    </filter>
```

```
  </route>
</camelContext>
```

Choice

The `choice` processor is a conditional statement that is used to route incoming messages to alternative targets. The alternative targets are each enclosed in a `when` element, which takes a predicate argument. If the predicate is true, the current target is selected, otherwise processing proceeds to the next `when` element in the rule. For example, the following `choice()` processor directs incoming messages to either *`Target1`*, *`Target2`*, or *`Target3`*, depending on the values of the predicates:

```
<camelContext id="buildSimpleRouteWithChoice"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <choice>
      <when>
        <!-- First predicate -->
        <simple>header.foo = 'bar'</simple>
        <to uri="Target1"/>
      </when>
      <when>
        <!-- Second predicate -->
        <simple>header.foo = 'manchu'</simple>
        <to uri="Target2"/>
      </when>
      <otherwise>
        <to uri="Target3"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

Recipient list

If you want the messages from a source endpoint, *`SourceURL`*, to be sent to more than one target, there are two alternative approaches you can use. One approach is to include multiple `to` elements in the route, for example:

```
<camelContext id="staticRecipientList"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <to uri="Target1"/>
    <to uri="Target2"/>
```

```
    <to uri="Target3"/>
  </route>
</camelContext>
```

The alternative approach is to add a`recipientList` element, which takes a list of recipients as its argument (dynamic recipient list). The advantage of using the `recipientList` element is that the list of recipients can be calculated at runtime. For example, the following rule generates a recipient list by reading the contents of the `recipientListHeader` from the incoming message:

```
<camelContext id="dynamicRecipientList"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <recipientList>
      <!-- Requires XPath 2.0 -->

<xpath>tokenize(/headers/recipientListHeader,"\s+")</xpath>
    </recipientList>
  </route>
</camelContext>
```

Splitter

The `splitter` processor is used to split a message into parts, which are then processed as separate messages. The `splitter` element must contain an expression that returns a list, where each item in the list represents a message part that is to be re-sent as a separate message. For example, the following rule splits the body of an incoming message into separate sections (represented by a top-level `section` element) and then sends each section to the target in a separate message:

```
<camelContext id="splitterRoute"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <splitter>
      <xpath>/section</xpath>
      <to uri="seda:b"/>
    </splitter>
```

```
  </route>
</camelContext>
```

Aggregator

The `aggregator` processor is used to aggregate related incoming messages into a single message. In order to distinguish which messages are eligible to be aggregated together, you need to define a *correlation key* for the aggregator. The correlation key is normally derived from a field in the message (for example, a header field). Messages that have the same correlation key value are eligible to be aggregated together. You can also optionally specify an aggregation algorithm to the `aggregator` processor (the default algorithm is to pck the latest message with a given value of the correlation key and to discard the older messages with that correlation key value).

For example, if you are monitoring a data stream that reports stock prices in real time, you might only be interested in the *latest* price of each stock symbol. In this case, you could configure an aggregator to transmit only the latest price for a given stock and discard the older (out-of-date) price notifications. The following rule implements this functionality, where the correlation key is read from the `stockSymbol` header and the default aggregator algorithm is used:

```
<camelContext id="aggregatorRoute"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <aggregator>
      <simple>header.stockSymbol</simple>
      <to uri="TargetURL"/>
    </aggregator>
  </route>
</camelContext>
```

Resequencer

A `resequencer` processor is used to re-arrange the order in which incoming messages are transmitted. The `resequencer` element needs to be provided with a sequence number (where the sequence number is calculated from the contents of a field in the incoming message). Naturally, before you can start re-ordering messages, you need to wait until a certain number of messages have been received from the source. There are a couple of different ways to specify how long the `resequencer` processor should wait before attempting to re-order the accumulated messages and forward them to the target, as follows:

- *Batch resequencing*—(the default) wait until a specified number of messages have accumulated before starting to re-order and forward messages. For example, the following resequencing rule would re-order messages based on the `timeOfDay` header, waiting until at least 300 messages have accumulated or 4000 ms have elapsed since the last message received.

```
<camelContext id="batchResequencer"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL" />
    <resequencer>
      <!-- Sequence ordering based on timeOfDay header -->
      <simple>header.timeOfDay</simple>
      <to uri="TargetURL" />
      <!--
        batch-config can be ommitted for default (batch)
resequencer settings
      -->
      <batch-config batchSize="300" batchTimeout="4000" />
    </resequencer>
  </route>
</camelContext>
```

- *Stream resequencing*—transmit messages as soon as they ariive *unless* the resequencer detects a gap in the incoming message stream (missing sequence numbers), in which case the resequencer waits until the missing messages arrive and then forwards the messages in the correct order. To avoid the resequencer blocking forever, you can specify a timeout (default is 1000 ms), after which time the message sequence is transmitted with unresolved gaps. For example, the following resequencing rule detects gaps in the message stream by monitoring the value of the `sequenceNumber` header, where the maximum buffer size is limited to 5000 and the timeout is specified to be 4000 ms:

```
<camelContext id="streamResequencer"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <resequencer>
      <simple>header.sequenceNumber</simple>
      <to uri="TargetURL" />
      <stream-config capacity="5000" timeout="4000"/>
    </resequencer>
```

```
  </route>
</camelContext>
```

Throttler

The `throttler` processor is used to ensure that a target endpoint does not get overloaded. The throttler works by limiting the number of messages that can pass through per second. If the incoming messages exceed the specified rate, the throttler accumulates excess messages in a buffer and transmits them more slowly to the target endpoint. For example, to limit the rate of throughput to 100 messages per second, you can define the following rule:

```
<camelContext id="throttlerRoute"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <throttler maximumRequestsPerPeriod="100"
timePeriodMillis="1000">
      <to uri="TargetURL"/>
    </throttler>
  </route>
</camelContext>
```

Delayer

The `delayer` processor is used to hold up messages for a specified length of time. The delay can either be relative (wait a specified length of time after receipt of the incoming message) or absolute (wait until a specific time). For example, to add a delay of 2 seconds before transmitting received messages, you can use the following rule:

```
<camelContext id="delayerRelative"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <delayer>
      <delay>2000</delay>
      <to uri="TargetURL"/>
    </delayer>
  </route>
</camelContext>
```

To wait until the absolute time specified in the `processAfter` header, you can use the following rule:

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

```
  <route>
    <from uri="SourceURL"/>
    <delayer>
      <simple>header.processAfter</simple>
      <to uri="TargetURL"/>
    </delayer>
  </route>
</camelContext>
```

Load balancer

The `loadBalance` processor is used to load balance message exchanges over a list of target endpoints. For example, to load balance incoming messages exchanges using a round robin algorithm (each endpoint in the target list is tried in sequence), you can use the following rule:

```
<camelContext id="loadBalancer"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <loadBalance>
      <to uri="TargetURL_01"/>
      <to uri="TargetURL_02"/>
      <roundRobin/>
    </loadBalance>
  </route>
</camelContext>
```

Currently, it is not possible to customize the load balancing algorithm in XML.

Languages for Expressions and Predicates

Overview

In the definition of a route, it is frequently necessary to evaluate expressions and predicates. For example, if a route includes a filter processor, you need to evaluate a predicate to determine whether or not a message is to be allowed through the filter. To facilitate the evaluation of expressions and predicates, Java Router supports multiple language plug-ins, which can be accessed through XML elements.

Elements for expressions and predicates

Table 5, "Elements for Expression and Predicate Languages" lists the elements that you can insert whenever the context demands an expression or a predicate. The content of the element must be a script written in the relevant language. At runtime, the return value of the script is read by the parent element.

Table 5. Elements for Expression and Predicate Languages

Element	Language	Description
`simple`	N/A	A simple expression language, native to Java Router (see Simple).
`xpath`	XPath	The XPath language, which is used to select element, attribute, and text nodes from XML documents (see http://www.w3schools.com/xpath/default.asp). The XPath expression is applied to the current message.
`xquery`	XQuery	The XQuery language, which is an extension of XPath (see http://www.w3schools.com/xquery/default.asp). The XQuery expression is applied to the current message.
`sql`	JoSQL	The JoSQL language, which is a language for extracting and manipulating data from collections of Java objects, using a SQL-like syntax (see http://josql.sourceforge.net/).
`ognl`	OGNL	The OGNL (Object Graph Navigation Language) language (see http://www.ognl.org/).

Element	Language	Description
`el`	EL	The Unified Expression Language (EL), originally developed as part of the JSP standard (see http://juel.sourceforge.net/).
`groovy`	Groovy	The Groovy scripting language (see http://groovy.codehaus.org/).
`javaScript`	JavaScript	The JavaScript scripting language (see http://developer.mozilla.org/en/docs/JavaScript), also known as ECMAScript (see http://www.ecmascript.org/).
`php`	PHP	The PHP scripting language (see http://www.php.net/).
`python`	Python	The Python scripting language (see http://www.python.org/).
`ruby`	Ruby	The Ruby scripting language (see http://www.ruby-lang.org/).
`bean`	Bean	Not really a language. The `bean` element is actually a mechanism for integrating with Java beans. You use the `bean` element to obtain an expression or predicate by invoking a method on a Java bean.

Transforming Message Content

Overview

This section describes how you can transform messages using the features provided in XML configuration.

Marshalling and unmarshalling

You can convert between low-level and high-level message formats using the following elements:

- `marshal`—convert a high-level data format to a low-level data format.
- `unmarshal`—convert a low-level data format to a high-level data format.

Java Router supports marshalling and unmarshalling of the following data formats:

- *Java serialization*—enables you to convert a Java object to a blob of binary data. For this data format, unmarshalling converts a binary blob to a Java object and marshalling converts a Java object to a binary blob. For example, to read a serialized Java object from an endpoint, *`SourceURL`*, and convert it to a Java object, you could use the following rule:

```
<camelContext id="serialization"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <serialization/>
    </unmarshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

- *JAXB*—provides a mapping between XML schema types and Java types (see https://jaxb.dev.java.net/). For JAXB, unmarshalling converts an XML data type to a Java object and marshalling converts a Java object to an XML data type. Before you can use JAXB data formats, you must compile your XML schema using a JAXB compiler in order to generate the Java classes that represent the XML data types in the schema. This is called *binding* the schema. After you have bound the schema, you can define a rule to unmarshal XML data to a Java object, as follows:

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

```
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <jaxb prettyPrint="true"
contextPath="GeneratedPackageName"/>
    </unmarshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

Where *GeneratedPackagename* is the name of the Java package generated by the JAXB compiler, which contains the Java classes representing your XML schema.

- *XMLBeans*—provides an alternative mapping between XML schema types and Java types (see http://xmlbeans.apache.org/). For XMLBeans, unmarshalling converts an XML data type to a Java object and marshalling converts a Java object to an XML data type. For example, to unmarshal XML data to a Java object using XMLBeans, define a rule like the following:

```
<camelContext id="xmlBeans"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <xmlBeans prettyPrint="true"/>
    </unmarshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

- *XStream*—is currently *not* supported in XML configuration.
- *Artix Data Services*—Java Router also integrates with Artix Data Services, enabling you to integrate stub code generated by Artix Data Services. See Artix Data Services for details.

Artix Data Services

Artix Data Services is a powerful tool for converting documents and messages between different data formats. In Artix Data Services, you can use a graphcial tool to define complex mapping rules (including processing of data content) and then generate stub code to implement the mapping rules (see IONA Artix Data Services [http://www.iona.com/products/artix/data_services.htm?WT.mc_id=125795] home page and the Artix Data Services documentation

[http://www.iona.com/support/docs/artix/data_services/3.6/index.xml] for more details). The `marshal` and `unmarshal` elements are capable of consuming Artix Data Services stub code in order to perform transformations on message formats. For example, Example 12, “Using Artix Data Services to Marshal and Unmarshal” shows a rule that unmarshals XML documents into a canonical format (Java objects) and then marshals the canonical format into the tag/value pair format.

Example 12. Using Artix Data Services to Marshal and Unmarshal

```
<camelContext id="artixDS"
xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <artixDS contentType="Xml"
elementTypeName="iso.std.iso.x20022.tech.xsd.pacs.x008.x001.x01.DocumentElement"/>

    </unmarshal>
    <marshal>
      <artixDS contentType="TagValuePair"/>
    </marshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

Where the `contentType` attribute can be set to one of the following values: `TagValuePair`, `Sax`, `Xml`, `Java`, `Text`, `Binary`, `Auto`, `Default`.

Note

Artix Data Services is licensed separately from Java Router.