Red Hat JBoss Fuse 6.0

Web Services and Routing with Camel CXF

Easy Web services with Apache Camel's CXF component

Last Updated: 2017-10-13

Red Hat JBoss Fuse 6.0 Web Services and Routing with Camel CXF

Easy Web services with Apache Camel's CXF component

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

This guide describes how to use Apache Camel's CXF component to create Web services or wrap existing functionality in Web service facades.

Table of Contents

CHAPTER 1. DEMONSTRATION CODE FOR CAMEL/CXF

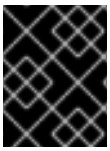
Abstract

This chapter explains how to install, build, and run the demonstrations that accompany this guide.

1.1. DOWNLOADING AND INSTALLING THE DEMONSTRATIONS

Overview

Most of the examples discussed in this guide are based on working demonstrations, which you can download and try out for yourself. The examples can easily be run by deploying them into a Red Hat JBoss Fuse container, as described here.



IMPORTANT

The demonstrations accompanying this guide are available to *subscription customers only*.

Prerequisites

For building and running the demonstration code, you must have the following prerequisites installed:

- *Java platform*—the demonstrations must run on the Java 6 platform from Oracle.
- *Apache Maven build tool*—to build the demonstration, you require Apache Maven 3.0.x (or Maven 2.2.1).
- *Internet connection*—Maven requires an Internet connection in order to download required dependencies from remote repositories while performing a build.
- *Red Hat JBoss Fuse*—the demonstrations are deployed into the JBoss Fuse container.

Downloading the demonstration package

The source code for the demonstrations is packaged as a Zip file, **`cxf-webinars-assembly-1.1.4-src.zip`**, and is available from the following location:

- cxf-webinars-assembly-1.1.4-src.zip

Installing the package

To install the package, simply extract the Zip archive into any convenient location on your file system.

1.2. RUNNING THE DEMONSTRATIONS

Building the demonstrations

Use Apache Maven to build the demonstrations. Open a new command prompt, change directory to ***`DemoDir`*`/src/fuse-webinars/cxf-webinars`**, and enter the following command:

```
mvn install
```

This command builds all of the demonstrations under the **cxf-webinars** directory (where the demonstrations are defined to be submodules of the **cxf-webinars/pom.xml** project). While Maven is building the demonstration code, it downloads whatever dependencies it needs from the Internet and installs them in the local Maven repository.

Starting and configuring the Red Hat JBoss Fuse container

Start and configure the Red Hat JBoss Fuse container as follows:

1. *(Optional)* If your local Maven repository is in a non-standard location, you might need to edit the JBoss Fuse configuration to specify your custom location. Edit the ***InstallDir*/etc/org.ops4j.pax.url.mvn.cfg** file and set the **org.ops4j.pax.url.mvn.localRepository** property to the location of your local Maven repository:

   ```
   #
   # Path to the local maven repository which is used to avoid
   downloading
   # artifacts when they already exist locally.
   # The value of this property will be extracted from the settings.xml
   file
   # above, or defaulted to:
   #     System.getProperty( "user.home" ) + "/.m2/repository"
   #
   #org.ops4j.pax.url.mvn.localRepository=
   org.ops4j.pax.url.mvn.localRepository=file:E:/Data/.m2/repository
   ```

2. Launch the JBoss Fuse container. Open a new command prompt, change directory to ***InstallDir*/bin**, and enter the following command:

   ```
   ./fuse
   ```

3. For convenience, each of the demonstrations can be deployed into the JBoss Fuse container as an Apache Karaf feature (which automatically installs any required dependencies along with the demonstration bundle). But first, you must specify the location of the features repository, by entering the following console command:

   ```
   JBossFuse:karaf@root> features:addUrl
   mvn:org.fusesource.sparks.fuse-webinars.cxf-webinars/customer-
   features/Version/xml
   ```

   Where *Version* is the current version of the demonstration package (see the value of the **project/version** element in the ***DemoDir*/src/pom.xml** file).

Demonstration features

The following features are now available from the JBoss Fuse console (where you can enter the command, **features:list | grep customer** to check the status of these features):

```
customer-ws
customer-ws-client
```

```
customer-ws-cxf-payload
customer-ws-cxf-pojo
customer-ws-cxf-provider
```

Running the customer-ws-osgi-bundle demonstration

It is now a relatively straightforward task to run each of the demonstrations by installing the relevant features.

For example, to start up the WSDL-first Web service (discussed in Chapter 3, *WSDL-First Service Implementation*), enter the following console command:

```
JBossFuse:karaf@root> features:install customer-ws
```

To see the Web service in action, start up the sample Web service client (discussed in Chapter 4, *Implementing a WS Client*), by entering the following console command:

```
JBossFuse:karaf@root> features:install customer-ws-client
```

The bundle creates a thread that invokes the Web service once a second and logs the response. View the log by entering the following console command:

```
JBossFuse:karaf@root> log:tail -n 4
```

You should see log output like the following:

```
18:03:58,609 | INFO  | qtp5581640-231   | CustomerServiceImpl
| ?                                   ? |
 218 - org.fusesource.sparks.fuse-webinars.cxf-webinars.customer-ws-osgi-
bundle - 1.1.4 | Getting status for custome
 r 1234
18:03:58,687 | INFO  |  invoker thread. | ClientInvoker
| ?                                   ? |
 219 - org.fusesource.sparks.fuse-webinars.cxf-webinars.customer-ws-client
- 1.1.4 | Got back: status = Active, stat
 usMessage = In the park, playing with my frisbee.
18:04:00,687 | INFO  | qtp5581640-232   | CustomerServiceImpl
| ?                                   ? |
 218 - org.fusesource.sparks.fuse-webinars.cxf-webinars.customer-ws-osgi-
bundle - 1.1.4 | Getting status for custome
 r 1234
18:04:00,703 | INFO  |  invoker thread. | ClientInvoker
| ?                                   ? |
 219 - org.fusesource.sparks.fuse-webinars.cxf-webinars.customer-ws-client
- 1.1.4 | Got back: status = Active, stat
 usMessage = In the park, playing with my frisbee.
```

To stop viewing the log, type the interrupt character (usually Ctrl-C).

To stop the client, first discover the client's bundle ID using the **`osgi:list`** console command. For example:

```
JBossFuse:karaf@root> list | grep customer-ws-client
[ 219] [Active     ] [            ] [Started] [   60] customer-ws-client
```

```
(1.1.4)
```

You can then stop the client using the **osgi:stop** console command. For example:

```
JBossFuse:karaf@root> stop 219
```

To shut down the container completely, enter the following console command:

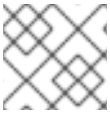
```
JBossFuse:karaf@root> shutdown -f
```

CHAPTER 2. JAVA-FIRST SERVICE IMPLEMENTATION

2.1. JAVA-FIRST OVERVIEW

Overview

The Java-first approach is a convenient way to get started with Web services, if you are unfamiliar with WSDL syntax. Using this approach, you can define the Web service interface using an ordinary Java interface and then use the provided Apache CXF utilities to generate the corresponding WSDL contract from the Java interface.



NOTE

There is no demonstration code to accompany this example.

Service Endpoint Interface (SEI)

An SEI is an ordinary Java interface. In order to use the standard JAX-WS frontend, the SEI must be annotated with the **`@WebService`** annotation.[1]

In the Java-first approach, the SEI is the starting point for implementing the Web service and it plays a central role in the development of the Web service implementation. The SEI is used in the following ways:

- *Base type of the Web service implementation (server side)*—you define the Web service by implementing the SEI.
- *Proxy type (client side)*—on the client side, you use the SEI to invoke operations on the client proxy object.
- *Basis for generating the WSDL contract*—in the Java-first approach, you generate the WSDL contract by converting the SEI to WSDL.

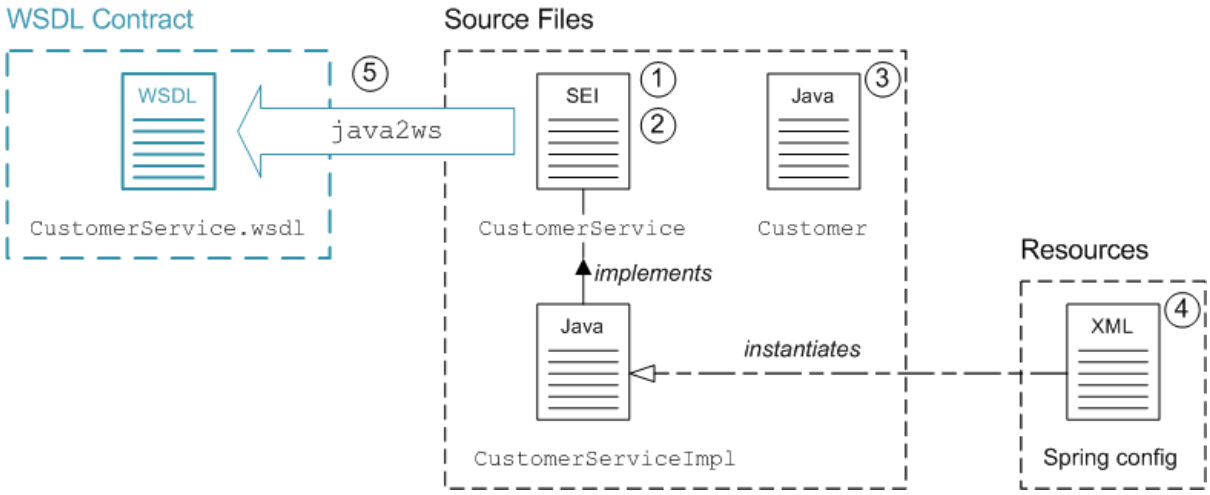
WSDL contract

The WSDL contract is a platform-neutral and language-neutral description of the Web service interface. When you want to make the Web service available to third-party clients, you should publish the WSDL contract to some well-known location. The WSDL contract contains all of the metadata required by WS clients.

The CustomerService demonstration

Figure 2.1, “Building a Java-First Web Service” shows an overview of the files required to implement and build the **`CustomerService`** Web service using the Java-first approach.

Figure 2.1. Building a Java-First Web Service



Implementing and building the service

To implement and build the Java-first example shown in Figure 2.1, “Building a Java-First Web Service”, you would perform the following steps:

1. Implement the SEI, which constitutes the basic definition of the Web service's interface.
2. Annotate the SEI (you can use the annotations to influence the ultimate form of the generated WSDL contract).
3. Implement any other requisite Java classes. In particular, implement the following:
   - Any data types referenced by the SEI—for example, the **`Customer`** class.
   - The implementation of the SEI, **`CustomerServiceImpl`**.
4. Instantiate the Web service endpoint, by adding the appropriate code to a Spring XML file.
5. Generate the WSDL contract using a Java-to-WSDL converter.

2.2. DEFINE SEI AND RELATED CLASSES

Overview

The Service Endpoint Interface (SEI) is the starting point for implementing a Web service in the Java-first approach. The SEI represents the Web service in Java and it is ultimately used as the basis for generating the WSDL contract. This section describes how to create a sample SEI, the **`CustomerService`** interface, which enables you to access the details of a customer's account.

The CustomerService SEI

A JAX-WS service endpoint interface (SEI) is essentially an ordinary Java interface, augmented by certain annotations (which are discussed in the next section). For example, consider the following **`CustomerService`** interface, which defines methods for accessing the **`Customer`** data type:

```
// Java
package com.fusesource.demo.wsdl.customerservice;
```

```
// NOT YET ANNOTATED!
public interface CustomerService {

    public com.fusesource.demo.customer.Customer lookupCustomer(
        java.lang.String customerId
    );

    public void updateCustomer(
        com.fusesource.demo.customer.Customer cust
    );

    public void getCustomerStatus(
        java.lang.String customerId,
        javax.xml.ws.Holder<java.lang.String> status,
        javax.xml.ws.Holder<java.lang.String> statusMessage
    );
}
```

After adding the requisite annotations to the **CustomerService** interface, this interface provides the basis for defining the **CustomerService** Web service.

javax.xml.ws.Holder<?> types

The **getCustomerStatus** method from the **CustomerService** interface has parameters declared to be of **javax.xml.ws.Holder<String>** type. These so-called *holder types* are needed in order to declare the **OUT** or **INOUT** parameters of a WSDL operation.

The syntax of WSDL operations allows you to define any number of OUT or INOUT parameters, which means that the parameters are used to *return* a value to the caller. This kind of parameter passing is *not* natively supported by the Java language. Normally, the only way that Java allows you to return a value is by declaring it as the return value of a method. You can work around this language limitation, however, by declaring parameters to be *holder* types.

For example, consider the definition of the following method, **getStringValues()**, which takes a holder type as its second parameter:

```
// Java
public void getStringValues(
    String wrongWay,
    javax.xml.ws.Holder<String> rightWay
) {
    wrongWay = "Caller will never see this string!";
    rightWay.value = "But the caller *can* see this string.";
}
```

The caller can access the value of the returned **rightWay** string as **rightWay.value**. For example:

```
// Java
String wrongWay = "This string never changes";
javax.xml.ws.Holder<String> rightWay.value =  "This value *can* change.";

sampleObject.getStringValues(wrongWay, rightWay);

System.out.println("Unchanged string: " + wrongWay);
System.out.println("Changed string: " + rightWay.value);
```

-

It is, perhaps, slightly unnatural to use **`Holder<>`** types in a Java-first example, because this is not a normal Java idiom. But it is interesting to include OUT parameters in the example, so that you can see how a Web service processes this kind of parameter.

Related classes

When you run the Java-to-WSDL compiler on the SEI, it converts not only the SEI, but also the classes referenced as parameters or return values. The parameter types must be convertible to XML, otherwise it would not be possible for WSDL operations to send or to receive those data types. In fact, when you run the Java-to-WSDL compiler, it is typically necessary to convert an entire tree of related classes to XML using the standard JAX-B encoding.

Normally, as long as the related classes do not require any exotic language features, the JAX-B encoding should be quite straightforward.

Default constructor for related classes

There is one simple rule, however, that you need to keep in mind when implementing related classes: each related class *must* have a default constructor (that is, a constructor without arguments). If you do not define any constructor for a class, the Java language automatically adds a default constructor. But if you define a class's constructors explicitly, you must ensure that one of them is a default constructor.

The Customer class

For example, the **`Customer`** class appears as a related class in the definition of the **`CustomerService`** SEI (the section called “The CustomerService SEI”). The **`Customer`** class consists of a collection of **`String`** fields and the only special condition it needs to satisfy is that it includes a default constructor:

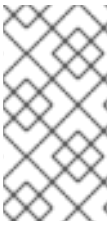
```
// Java
package com.fusesource.demo.customer;

public class Customer {
    protected String firstName;
    protected String lastName;
    protected String phoneNumber;
    protected String id;

    // Default constructor, required by JAX-WS
 public Customer() { }

    public Customer(String firstName, String lastName, String phoneNumber,
            String id) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
        this.phoneNumber = phoneNumber;
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String value) {
```

```
        this.firstName = value;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String value) {
        this.lastName = value;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String value) {
        this.phoneNumber = value;
    }

    public String getId() {
        return id;
    }

    public void setId(String value) {
        this.id = value;
    }
}
```

2.3. ANNOTATE SEI FOR JAX-WS

Overview

To use the JAX-WS frontend, an SEI must be annotated using standardised JAX-WS annotations. The annotations signal to the Web services tooling that the SEI uses JAX-WS and the annotations are also used to customize the mapping from Java to WSDL. Here we only cover the most basic annotations—for complete details of JAX-WS annotations, see *Developing Applications Using JAX-WS* from the Apache CXF library.



NOTE

It sometimes makes sense also to annotate the service implementation class (the class that implements the SEI)—for example, if you want to associate an implementation class with a specific WSDL **`serviceName`** and **`portName`** (there can be more than one implementation of a given SEI).

Minimal annotation

The minimal annotation required for an SEI using the JAX-WS frontend is to prefix the interface declaration with **`@WebService`**. For example, the **`CustomerService`** SEI could be minimally annotated as follows:

```
// Java
package com.fusesource.demo.wsdl.customerservice;
```

```
import javax.jws.WebService;

@WebService
public interface CustomerService {
  ...
}
```

If you run the Java-to-WSDL utility on this interface, it will generate a complete WSDL contract using the standard default style of code conversion.

@WebService annotation

Although it is sufficient to specify the **@WebService** annotation without any attributes, it is usually better to specify some attributes to provide a more descriptive WSDL service name and WSDL port name. You will also usually want to specify the XML target namespace. For this, you can specify the following optional attributes of the **@WebService** annotation:

name

Specifies the name of the WSDL contract (appearing in the **wsdl:definitions** element).

serviceName

Specifies the name of the WSDL service (a SOAP service is defined by default in the generated contract).

portName

Specifies the name of the WSDL port (a SOAP/HTTP port is defined by default in the generated contract).

targetNamespace

The XML schema namespace that is used, by default, to qualify the elements and types defined in the contract.

@WebParam annotation

You can add the **@WebParam** annotation to method arguments in the SEI. The **@WebParam** annotation is optional, but there are a couple of good reasons for adding it:

- By default, JAX-WS maps Java arguments to parameters with names like **arg0**, ..., **arg*N***. Messages are much easier to read, however, when the parameters have meaningful names.
- It is a good idea to define parameter elements without a namespace. This makes the XML encoding of requests and responses more compact.
- To enable support for WSDL OUT and INOUT parameters.

You can add **@WebParam** annotations with the following attributes:

name

Specifies the mapped name of the parameter.

targetNamespace

Specifies the namespace of the mapped parameter. Set this to a blank string for a more compact XML encoding.

mode

Can have one of the following values:

- **WebParam.Mode.IN**—*(default)* parameter is passed from client to service (in request).
- **WebParam.Mode.INOUT**—parameter is passed from client to service (request) and from the service back to the client (in reply).
- **WebParam.Mode.OUT**—parameter is passed from service back to the client (in reply).

OUT and INOUT parameters

In WSDL, OUT and INOUT parameters represent values that can be sent from the service back to the client (where the INOUT parameter is sent in both directions).

In Java syntax, the only value that can ordinarily be returned from a method is the method's return value. In order to support OUT or INOUT parameters in Java (which are effectively like additional return values), you must:

- Declare the corresponding Java argument using a **javax.xml.ws.Holder<*ParamType*>** type, where ***ParamType*** is the type of the parameter you want to send.
- Annotate the Java argument with **@WebParam**, setting either **mode = WebParam.Mode.OUT** or **mode = WebParam.Mode.INOUT**.

Annotated CustomerService SEI

The following example shows the **CustomerService** SEI after it has been annotated. Many other annotations are possible, but this level of annotation is usually adequate for a WSDL-first project.

```
// Java
package com.fusesource.demo.wsdl.customerservice;

import javax.jws.WebParam;
import javax.jws.WebService;

@WebService(
  targetNamespace = "http://demo.fusesource.com/wsdl/CustomerService/",
  name = "CustomerService",
  serviceName = "CustomerService",
  portName = "SOAPOverHTTP"
)
public interface CustomerService {

    public com.fusesource.demo.customer.Customer lookupCustomer(
 @WebParam(name = "customerId", targetNamespace = "")
        java.lang.String customerId
    );

    public void updateCustomer(
 @WebParam(name = "cust", targetNamespace = "")
```

```
        com.fusesource.demo.customer.Customer cust
    );

    public void getCustomerStatus(
 @WebParam(name = "customerId", targetNamespace = "")
        java.lang.String customerId,
 @WebParam(mode = WebParam.Mode.OUT, name = "status", targetNamespace =
"")
        javax.xml.ws.Holder<java.lang.String> status,
 @WebParam(mode = WebParam.Mode.OUT, name = "statusMessage",
targetNamespace = "")
        javax.xml.ws.Holder<java.lang.String> statusMessage
    );
}
```

2.4. INSTANTIATE THE WS ENDPOINT

Overview

In Apache CXF, you create a WS endpoint by defining a **jaxws:endpoint** element in XML. The WS endpoint is effectively the runtime representation of the Web service: it opens an IP port to listen for SOAP/HTTP requests, is responsible for marshalling and unmarshalling messages (making use of the generated Java stub code), and routes incoming requests to the relevant methods on the implementor class.

In other words, creating a Web service in Spring XML consists essentially of the following two steps:

1. Create an instance of the implementor class, using the Spring **bean** element.
2. Create a WS endpoint, using the **jaxws:endpoint** element.

The jaxws:endpoint element

You can instantiate a WS endpoint using the **jaxws:endpoint** element in a Spring file, where the **jaxws:** prefix is associated with the **http://cxf.apache.org/jaxws** namespace.

NOTE

Take care not to confuse the **jaxws:endpoint** element with the **cxf:cxfEndpoint** element, which you meet later in this guide: the **jaxws:endpoint** element is used to integrate a WS endpoint with a Java implementation class; whereas the **cxf:cxfEndpoint** is used to integrate a WS endpoint with a Camel route.

Define JAX-WS endpoint in XML

The following sample Spring file shows how to define a JAX-WS endpoint in XML, using the **jaxws:endpoint** element.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jaxws="http://cxf.apache.org/jaxws"
      xmlns:soap="http://cxf.apache.org/bindings/soap"
```

```
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/bindings/soap
http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

  <jaxws:endpoint
    xmlns:customer="http://demo.fusesource.com/wsdl/CustomerService/"
    id="customerService"
    address="/Customer"
    serviceName="customer:CustomerService"
    endpointName="customer:SOAPOverHTTP"
    implementor="#customerServiceImpl">
  </jaxws:endpoint>

  <bean id="customerServiceImpl"
      class="com.fusesource.customer.ws.CustomerServiceImpl"/>

</beans>
```

Address for the Jetty container

Apache CXF deploys the WS endpoint into a Jetty servlet container instance and the **`address`** attribute of **`jaxws:endpoint`** is therefore used to configure the addressing information for the endpoint in the Jetty container.

Apache CXF supports the notion of a *default servlet container* instance. The way the default servlet container is initialized and configured depends on the particular mode of deployment that you choose. For example the Red Hat JBoss Fuse container and Web containers (such as Tomcat) provide a default servlet container.

There are two different syntaxes you can use for the endpoint address, where the syntax that you use effectively determines whether or not the endpoint is deployed into the default servlet container, as follows:

- *Address syntax for default servlet container*—to use the default servlet container, specify only the servlet context for this endpoint. Do *not* specify the protocol, host, and IP port in the address. For example, to deploy the endpoint to the **`/Customers`** servlet context in the default servlet container:

  ```
  address="/Customers"
  ```

- *Address syntax for custom servlet container*—to instantiate a custom Jetty container for the endpoint, specify a complete HTTP URL, including the host and IP port (the value of the IP port effectively identifies the target Jetty container). Typically, for a Jetty container, you specify the host as **`0.0.0.0`**, which is interpreted as a wildcard that matches every IP network interface on the local machine (that is, if deployed on a multi-homed host, Jetty opens a listening port on every network card). For example, to deploy the endpoint to the custom Jetty container listening on IP port, **`8083`**:

  ```
  address="http://0.0.0.0:8083/Customers"
  ```

NOTE

If you want to configure a secure endpoint (secured by SSL), you would specify the **`https:`** scheme in the address.

Referencing the service implementation

The **`implementor`** attribute of the **`jaxws:endpoint`** element references the implementation of the WS service. The value of this attribute can either be the name of the implementation class or (as in this example) a bean reference in the format, ***`#BeanID`***, where the # character indicates that the following identifier is the name of a bean in the bean registry.

2.5. JAVA-TO-WSDL MAVEN PLUG-IN

Overview

To generate a WSDL contract from your SEI, you can use either the **`java2ws`** command-line utility or the **`cxf-java2ws-plugin`** Maven plug-in. The plug-in approach is ideal for Maven-based projects: after you paste the requisite plug-in configuration into your POM file, the WSDL code generation step is integrated into your build.

Configure the Java-to-WSDL Maven plug-in

Configuring the Java-to-WSDL Maven plug-in is relatively easy, because most of the default configuration settings can be left as they are. After copying and pasting the sample **`plugin`** element into your project's POM file, there are just a few basic settings that need to be customized, as follows:

- *CXF version*—make sure that the plug-in's dependencies are using the latest version of Apache CXF.
- *SEI class name*—specify the fully-qualified class name of the SEI in the **`configuration/className`** element.
- *Location of output*—specify the location of the generated WSDL file in the **`configuration/outputFile`** element.

For example, the following POM fragment shows how to configure the **`cxf-java2ws-plugin`** plug-in to generate WSDL from the CustomerService SEI:

```
<project ...>
  ...
  <properties>
 <cxf.version>2.6.0.redhat-60024</cxf.version>
  </properties>

  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-java2ws-plugin</artifactId>
        <version>${cxf.version}</version>
        <dependencies>
```

```
          <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-frontend-jaxws</artifactId>
            <version>${cxf.version}</version>
          </dependency>
          <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-frontend-simple</artifactId>
            <version>${cxf.version}</version>
          </dependency>
        </dependencies>
        <executions>
          <execution>
            <id>process-classes</id>
            <phase>process-classes</phase>
            <configuration>

<className>org.fusesource.demo.camelcxf.ws.server.CustomerService</classNa
me>

<outputFile>${basedir}/../src/main/resources/wsdl/CustomerService.wsdl</ou
tputFile>
              <genWsdl>true</genWsdl>
              <verbose>true</verbose>
            </configuration>
            <goals>
              <goal>java2ws</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

</project>
```

Generated WSDL

When using the Java-first approach to defining a Web service, there are typically other parts of your application (for example, WS clients) that depend on the generated WSDL file. For this reason, it is generally a good idea to output the generated WSDL file to a common location, which is accessible to other projects in your application, using the **`outputFile`** configuration element.

If you do not specify the **`outputFile`** configuration element, the generated WSDL is sent to the following location, by default:

```
BaseDir/target/generated/wsdl/SEIClassName.wsdl
```

Reference

For full details of how to configure the Java-to-WSDL plug-in, see the Maven Java2WS plug-in reference page.

[1] If the SEI is left without annotations, Apache CXF defaults to using the simple frontend. This is a non-standard frontend, which is *not* recommended for most applications.

CHAPTER 3. WSDL-FIRST SERVICE IMPLEMENTATION

3.1. WSDL-FIRST OVERVIEW

Overview

If you are familiar with the syntax of WSDL and you want to have ultimate control over the layout and conventions applied to the WSDL contract, you will probably prefer to develop your Web service using the WSDL-first approach. In this approach, you start with the WSDL contract and then use the provided Apache CXF utilities to generate the requisite Java stub files from the WSDL contract.

Demonstration location

The code presented in this chapter is taken from the following demonstration:

> *DemoDir*/src/fuse-webinars/cxf-webinars/**customer-ws-osgi-bundle**

For details of how to download and install the demonstration code, see Chapter 1, *Demonstration Code for Camel/CXF*

WSDL contract

The WSDL contract is a platform-neutral and language-neutral description of the Web service interface. In the WSDL-first approach, the WSDL contract is the starting point for implementing the Web service. You can use it to generate Java stub code, which provides the basis for implementing the Web service on the server side.

Service Endpoint Interface (SEI)

The most important piece of the generated stub code is the SEI, which is an ordinary Java interface that represents the Web service interface in the Java language.

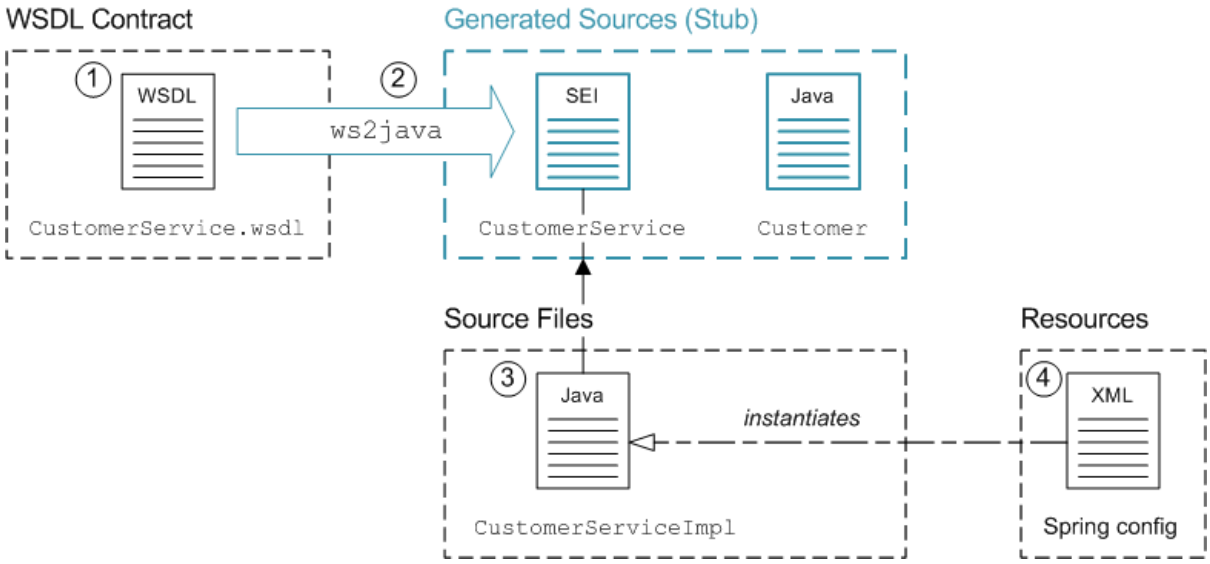
The SEI is used in the following ways:

- *Base type of the Web service implementation (server side)*—you define the Web service by implementing the SEI.
- *Proxy type (client side)*—on the client side, you use the SEI to invoke operations on the client proxy object.

The CustomerService demonstration

Figure 3.1, “Building a WSDL-First Web Service” shows an overview of the files required to implement and build the **CustomerService** Web service using the WSDL-first approach.

Figure 3.1. Building a WSDL-First Web Service



Implementing and building the service

To implement and build the WSDL-first example shown in Figure 3.1, “Building a WSDL-First Web Service”, starting from scratch, you would perform the following steps:

1. Create the WSDL contract.
2. Generate the Java stub code from the WSDL contract using a WSDL-to-Java converter, **`ws2java`**. This gives you the SEI, **`CustomerService`**, and its related classes, such as **`Customer`**.
3. Write the implementation of the SEI, **`CustomerServiceImpl`**.
4. Instantiate the Web service endpoint, by adding the appropriate code to a Spring XML file.

3.2. CUSTOMERSERVICE WSDL CONTRACT

Sample WSDL contract

The WSDL contract used in this demonstration is the **`CustomerService`** WSDL contract, which is available in the following location:

```
/fuse-webinars/cxf-webinars/src/main/resources
```

Because the WSDL contract is a fairly verbose format, it is not shown in here in full. The main point you need to be aware of is that the **`CustomerSerivice`** WSDL contract exposes the following operations:

`lookupCustomer`

Given a customer ID, the operation returns the corresponding **`Customer`** data object.

`updateCustomer`

Stores the given **`Customer`** data object against the given customer ID.

`getCustomerStatus`

Returns the status of the customer with the given customer ID.

Parts of the WSDL contract

A WSDL contract has the following main parts:

- the section called “Port type”.
- the section called “WSDL binding”.
- the section called “WSDL port”.

Port type

The port type is defined in the WSDL contract by the **`wsdl:portType`** element. It is analogous to an interface and it defines the operations that can be invoked on the Web service.

For example, the following WSDL fragment shows the **`wsdl:portType`** definition from the **`CustomerService`** WSDL contract:

```
<wsdl:definitions name="CustomerService"
 targetNamespace="http://demo.fusesource.com/wsdl/CustomerService/"
    ...>

    ...
    <wsdl:portType name="CustomerService">
        <wsdl:operation name="lookupCustomer">
            <wsdl:input message="tns:lookupCustomer"></wsdl:input>
            <wsdl:output message="tns:lookupCustomerResponse">
</wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="updateCustomer">
            <wsdl:input message="tns:updateCustomer"></wsdl:input>
            <wsdl:output message="tns:updateCustomerResponse">
</wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="getCustomerStatus">
            <wsdl:input message="tns:getCustomerStatus"></wsdl:input>
            <wsdl:output message="tns:getCustomerStatusResponse">
</wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
    ...
</wsdl:definitions>
```

WSDL binding

A WSDL binding describes how to *encode* all of the operations and data types associated with a particular port type. A binding is specific to a particular protocol—for example, SOAP or JMS.

WSDL port

A WSDL port *specifies the transport protocol* and contains addressing data that enables clients to locate and connect to a remote server endpoint.

For example, the **`CustomerService`** WSDL contract defines the following WSDL port:

```
<wsdl:definitions ...>
    ...
    <wsdl:service name="CustomerService">
        <wsdl:port name="SOAPOverHTTP" binding="tns:CustomerServiceSOAP">
            <soap:address location="http://0.0.0.0:8183/CustomerService"
/>
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

The address specified by the **`soap:address`** element's **`location`** attribute in the original WSDL contract is typically overridden at run time, however.

The getCustomerStatus operation

Because a WSDL contract is fairly verbose, it can be a bit difficult to see what the parameters of an operation are. Typically, for each operation, you can find data types in the XML schema section that represent the operation request and the operation response. For example, the **`getCustomerStatus`** operation has its request parameters (IN parameters) encoded by the **`getCustomerStatus`** element and its response parameters (OUT parameters) encoded by the **`getCustomerStatusResponse`** element, as follows:

```
<wsdl:definitions name="CustomerService"
 targetNamespace="http://demo.fusesource.com/wsdl/CustomerService/"
    ...>
    <wsdl:types>
        <xsd:schema ...>
            ...
            <xsd:element name="getCustomerStatus">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="customerId"
type="xsd:string"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="getCustomerStatusResponse">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="status" type="xsd:string"/>
                        <xsd:element name="statusMessage"
type="xsd:string"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </wsdl:types>
    ...
</wsdl:definitions>
```

References

For more details about the format of WSDL contracts and how to create your own WSDL contracts, see *Writing WSDL Contracts* and the Eclipse JAX-WS Tools Component.

3.3. WSDL-TO-JAVA MAVEN PLUG-IN

Overview

In contrast to the Java-first approach, which starts with a Java interface and then generates the WSDL contract, the WSDL-first approach needs to generate Java stub code from the WSDL contract.

To generate Java stub code from the WSDL contract, you can use either the **`ws2java`** command-line utility or the **`cxf-codegen-plugin`** Maven plug-in. The plug-in approach is ideal for Maven-based projects: after you paste the requisite plug-in configuration into your POM file, the WSDL-to-Java code generation step is integrated into your build.

Configure the WSDL-to-Java Maven plug-in

Configuring the WSDL-to-Java Maven plug-in is relatively easy, because most of the default configuration settings can be left as they are. After copying and pasting the sample **`plugin`** element into your project's POM file, there are just a few basic settings that need to be customized, as follows:

- *CXF version*—make sure that the plug-in's dependencies are using the latest version of Apache CXF.
- *WSDL file location*—specify the WSDL file location in the **`configuration/wsdlOptions/wsdlOption/wsdl`** element.
- *Location of output*—specify the root directory of the generated Java source files in the **`configuration/sourceRoot`** element.

For example, the following POM fragment shows how to configure the **`cxf-codegen-plugin`** plug-in to generate Java stub code from the **`CustomerService.wsdl`** WSDL file:

```
<project ...>
  ...
  <properties>
    <cxf.version>2.6.0.redhat-60024</cxf.version>
  </properties>

  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-codegen-plugin</artifactId>
        <version>${cxf.version}</version>
        <executions>
          <execution>
            <id>generate-sources</id>
            <phase>generate-sources</phase>
            <configuration>
```

```
              <!-- Maven auto-compiles any source files under
target/generated-sources/ -->
              <sourceRoot>${basedir}/target/generated-
sources/jaxws</sourceRoot>
              <wsdlOptions>
                <wsdlOption>

<wsdl>${basedir}/../src/main/resources/wsdl/CustomerService.wsdl</wsdl>
                </wsdlOption>
              </wsdlOptions>
            </configuration>
            <goals>
              <goal>wsdl2java</goal>
            </goals>
          </execution>
        </executions>
      </plugin>

    </plugins>
  </build>

</project>
```

Generated Java source code

With the sample configuration shown here, the generated Java source code is written under the **`target/generated-sources/jaxws`** directory. Note that the Web service implementation is dependent on this generated stub code—for example, the service implementation class must implement the generated **`CustomerService`** SEI.

Adding the generated source to an IDE

If you are using an IDE such as Eclipse or Intellij's IDEA, you need to make sure that the IDE is aware of the generated Java code. For example, in Eclipse it is necessary to add the **`target/generated-sources/jaxws`** directory to the project as a source code directory.

Compiling the generated code

You must ensure that the generated Java code is compiled and added to the deployment package. By convention, Maven automatically compiles any source files that it finds under the following directory:

```
BaseDir/target/generated-sources/
```

Hence, if you configure the output directory as shown in the preceding POM fragment, the generated code is automatically compiled by Maven.

Reference

For full details of how to configure the Java-to-WSDL plug-in, see the Maven cxf-codegen-plugin reference page.

3.4. INSTANTIATE THE WS ENDPOINT

Overview

In Apache CXF, you create a WS endpoint by defining a **jaxws:endpoint** element in XML. The WS endpoint is effectively the runtime representation of the Web service: it opens an IP port to listen for SOAP/HTTP requests, is responsible for marshalling and unmarshalling messages (making use of the generated Java stub code), and routes incoming requests to the relevant methods on the implementor class.

In other words, creating a Web service in Spring XML consists essentially of the following two steps:

1. Create an instance of the implementor class, using the Spring **bean** element.
2. Create a WS endpoint, using the **jaxws:endpoint** element.

Define JAX-WS endpoint in XML

The following sample Spring file shows how to define a JAX-WS endpoint in XML, using the **jaxws:endpoint** element.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jaxws="http://cxf.apache.org/jaxws"
      xmlns:soap="http://cxf.apache.org/bindings/soap"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/bindings/soap
http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

  <jaxws:endpoint
    xmlns:customer="http://demo.fusesource.com/wsdl/CustomerService/"
    id="customerService"
    address="/Customer"
    serviceName="customer:CustomerService"
    endpointName="customer:SOAPOverHTTP"
    implementor="#customerServiceImpl">
  </jaxws:endpoint>

  <bean id="customerServiceImpl"
      class="com.fusesource.customer.ws.CustomerServiceImpl"/>

</beans>
```

Address for the Jetty container

In the preceding example, the **address** attribute of the **jaxws:endpoint** element specifies the servlet context for this endpoint, relative to the Jetty container in which it is deployed.

For more details about the options for specifying the endpoint address, see the section called “Address for the Jetty container”.

Referencing the service implementation

The **implementor** attribute of the **jaxws:endpoint** element is used to reference the implementation of the WS service. The value of this attribute can either be the name of the implementation class or (as in this example) a bean reference in the format, **#*BeanID***, where the # character indicates that the following identifier is the name of a bean in the bean registry.

3.5. DEPLOY TO AN OSGI CONTAINER

Overview

One of the options for deploying the Web service is to package it as an OSGi bundle and deploy it into an OSGi container such as Red Hat JBoss Fuse. Some of the advantages of an OSGi deployment include:

- Bundles are a relatively lightweight deployment option (because dependencies can be shared between deployed bundles).
- OSGi provides sophisticated dependency management, ensuring that only version-consistent dependencies are added to the bundle's classpath.

Using the Maven bundle plug-in

The Maven bundle plug-in is used to package your project as an OSGi bundle, in preparation for deployment into the OSGi container. There are two essential modifications to make to your project's **pom.xml** file:

1. Change the packaging type to **bundle** (by editing the value of the **project/packaging** element in the POM).
2. Add the Maven bundle plug-in to your POM file and configure it as appropriate.

Configuring the Maven bundle plug-in is quite a technical task (although the default settings are often adequate). For full details of how to customize the plug-in configuration, consult *Deploying into the OSGi Container* and *Managing OSGi Dependencies*.

Sample bundle plug-in configuration

The following POM fragment shows a sample configuration of the Maven bundle plug-in, which is appropriate for the current example.

```
<?xml version="1.0"?>
<project  ...>
  ...
  <groupId>org.fusesource.sparks.fuse-webinars.cxf-webinars</groupId>
  <artifactId>customer-ws-osgi-bundle</artifactId>
  <name>customer-ws-osgi-bundle</name>
  <url>http://www.fusesource.com</url>
 <packaging>bundle</packaging>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
```

```
        <version>${maven-bundle-plugin.version}</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Export-Package>
              !com.fusesource.customer.ws,
              !com.fusesource.demo.customer,
              !com.fusesource.demo.wsdl.customerservice
            </Export-Package>
            <Import-Package>
              META-INF.cxf,
              META-INF.cxf.osgi,
              *
            </Import-Package>
            <DynamicImport-Package>
              org.apache.cxf.*,
              org.springframework.beans.*
            </DynamicImport-Package>
          </instructions>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
</project>
```

Dynamic imports

The Java packages from Apache CXF and the Spring API are imported using dynamic imports (specified using the **`DynamicImport-Package`** element). This is a pragmatic way of dealing with the fact that Spring XML files are not terribly well integrated with the Maven bundle plug-in. At build time, the Maven bundle plug-in is *not* able to figure out which Java classes are required by the Spring XML code. By listing wildcarded package names in the **`DynamicImport-Package`** element, however, you allow the OSGi container to figure out which Java classes are needed by the Spring XML code at *run* time.

NOTE

In general, using **`DynamicImport-Package`** headers is not recommended in OSGi, because it short-circuits OSGi version checking. Normally, what should happen is that the Maven bundle plug-in lists the Java packages used at *build* time, along with their versions, in the **`Import-Package`** header. At *deploy* time, the OSGi container then checks that the available Java packages are compatible with the build-time versions listed in the **`Import-Package`** header. With dynamic imports, this version checking cannot be performed.

Build and deploy the service bundle

After you have configured the POM file, you can build the Maven project and install it in your local repository by entering the following command:

```
mvn install
```

To deploy the service bundle, enter the following command at the command console:

```
karaf@root> install -s mvn:org.fusesource.sparks.fuse-webinars.cxf-
webinars/customer-ws-osgi-bundle
```

NOTE

If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **`org.ops4j.pax.url.mvn.localRepository`** property in the ***`EsbInstallDir`*`/etc/org.ops4j.pax.url.mvn.cfg`** file, before you can use the **`mvn:`** scheme to access Maven artifacts.

Red Hat JBoss Fuse default servlet container

Red Hat JBoss Fuse has a default Jetty container which, by default, listens for HTTP requests on port 8181. Moreover, WS endpoints in this container are implicitly deployed under the servlet context **`cxf/`**. Hence, any WS endpoint whose **`address`** attribute is configured in the **`jaxws:endpoint`** element as ***`/EndpointContext`*** will have the following effective address:

```
http://Hostname:8181/cxf/EndpointContext
```

You can optionally customize the default servlet container by editing settings in the following file:

```
EsbInstallDir/etc/org.ops4j.pax.web.cfg
```

Full details of the properties you can set in this file are given in the Ops4j Pax Web configuration reference..

Check that the service is running

A simple way of checking that the service is running is to point your browser at the following URL:

```
http://localhost:8181/cxf/Customers?wsdl
```

This query should return a copy of the WS endpoint's WSDL contract.

CHAPTER 4. IMPLEMENTING A WS CLIENT

4.1. WS CLIENT OVERVIEW

Overview

The key object in a WS client is the WS client proxy object, which enables you to access the remote Web service by invoking methods on the SEI. The proxy object itself can easily be instantiated using the **`jaxws:client`** element in Spring XML.

Demonstration location

The code presented in this chapter is taken from the following demonstration:

```
DemoDir/src/fuse-webinars/cxf-webinars/customer-ws-client
```

For details of how to download and install the demonstration code, see Chapter 1, *Demonstration Code for Camel/CXF*

WSDL contract

The WSDL contract is a platform-neutral and language-neutral description of the Web service interface. It contains all of the metadata that a client needs to find a Web service and invoke its operations. You can generate Java stub code from the WSDL contract, which provides an API that makes it easy to invoke the remote WSDL operations.

Service Endpoint Interface (SEI)

The most important piece of the generated stub code is the SEI, which is an ordinary Java interface that represents the Web service interface in the Java language.

WS client proxy

The WS client proxy is an object that converts Java method invocations to remote procedure calls, sending and receiving messages to a remote instance of the Web service across the network. The methods of the proxy are exposed through the SEI.

NOTE

The proxy type is generated *dynamically* by Apache CXF at run time. That is, their is no class in the stub code that corresponds to the implementation of the proxy (the only relevant entity is the SEI, which defines the proxy's interface).

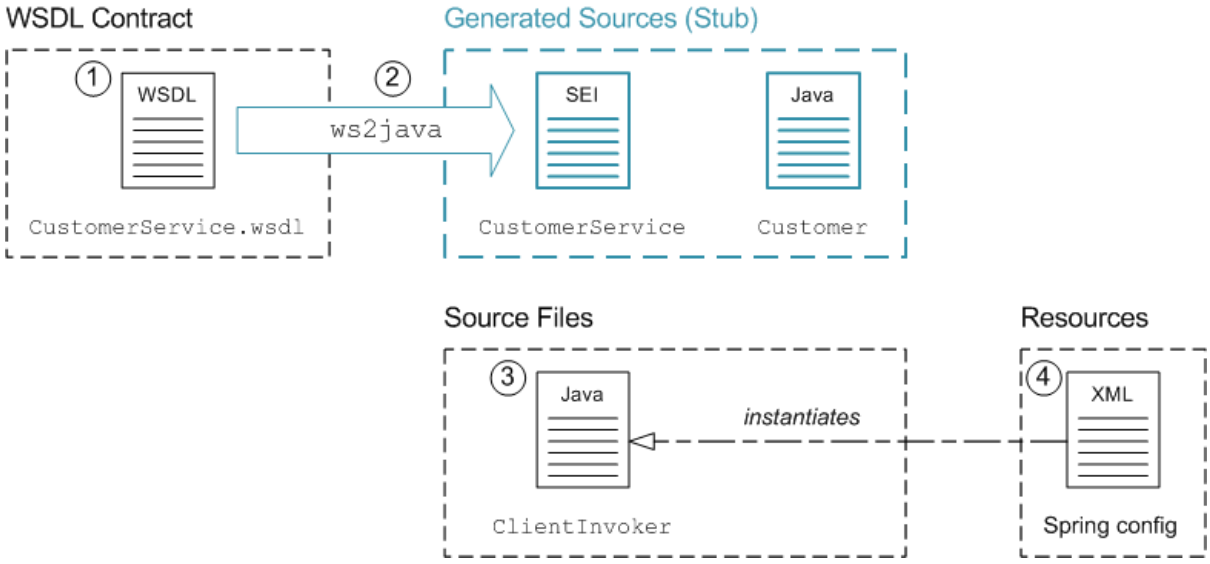
The CustomerService client

To take a specific example, consider the **`customer-ws-client`** demonstration, which is available from the following location:

```
DemoDir/src/fuse-webinars/cxf-webinars/customer-ws-client
```

Figure 4.1, “Building a WS Client” shows an overview of the files required to implement and build the WS client.

Figure 4.1. Building a WS Client



Implementing and building the WS client

To implement and build the sample WS client shown in Figure 4.1, “Building a WS Client”, starting from scratch, you would perform the following steps:

1. Obtain a copy of the WSDL contract.
2. Generate the Java stub code from the WSDL contract using a WSDL-to-Java converter, **`ws2java`**. This gives you the SEI, **`CustomerService`**, and its related classes, such as **`Customer`**.
3. Implement the main client class, **`ClientInvoker`**, which invokes the Web service operations. In this class define a bean property of type, **`CustomerService`**, so that the client class can receive a reference to the WS client proxy by property injection.
4. In a Spring XML file, instantiate the WS client proxy and inject it into the main client class, **`ClientInvoker`**.

4.2. WSDL-TO-JAVA MAVEN PLUG-IN

Overview

To generate Java stub code from the WSDL contract, you can use either the **`ws2java`** command-line utility or the **`cxf-codegen-plugin`** Maven plug-in. When using Maven, the plug-in approach is ideal: after you paste the requisite plug-in configuration into your POM file, the WSDL-to-Java code generation step is integrated into your build.

Configure the WSDL-to-Java Maven plug-in

Configuring the WSDL-to-Java Maven plug-in is relatively easy, because most of the default configuration settings can be left as they are. After copying and pasting the sample **`plugin`** element into your project's POM file, there are just a few basic settings that need to be customized, as follows:

- *CXF version*—make sure that the plug-in's dependencies are using the latest version of Apache CXF.
- *WSDL file location*—specify the WSDL file location in the **`configuration/wsdlOptions/wsdlOption/wsdl`** element.
- *Location of output*—specify the root directory of the generated Java source files in the **`configuration/sourceRoot`** element.

For example, the following POM fragment shows how to configure the **`cxf-codegen-plugin`** plug-in to generate Java stub code from the **`CustomerService.wsdl`** WSDL file:

```
<project ...>
  ...
  <properties>
    <cxf.version>2.6.0.redhat-60024</cxf.version>
  </properties>

  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-codegen-plugin</artifactId>
        <version>${cxf.version}</version>
        <executions>
          <execution>
            <id>generate-sources</id>
            <phase>generate-sources</phase>
            <configuration>
              <sourceRoot>${basedir}/target/generated-
sources/jaxws</sourceRoot>
              <wsdlOptions>
                <wsdlOption>

<wsdl>${basedir}/../src/main/resources/wsdl/CustomerService.wsdl</wsdl>
                </wsdlOption>
              </wsdlOptions>
            </configuration>
            <goals>
              <goal>wsdl2java</goal>
            </goals>
          </execution>
        </executions>
      </plugin>

    </plugins>
  </build>

</project>
```

Generated Java source code

With the sample configuration shown here, the generated Java source code is written under the

target/generated-sources/jaxws directory. Note that the client implementation is dependent on this generated stub code—for example, the client invokes the proxy using the generated **CustomerService** SEI.

Add generated source to IDE

If you are using an IDE such as Eclipse or Intellij's IDEA, you need to make sure that the IDE is aware of the generated Java code. For example, in Eclipse it is necessary to add the **target/generated-sources/jaxws** directory to the project as a source code directory.

Compiling the generated code

You must ensure that the generated Java code is compiled and added to the deployment package. By convention, Maven automatically compiles any source files that it finds under the following directory:

```
BaseDir/target/generated-sources/
```

Hence, if you configure the output directory as shown in the preceding POM fragment, the generated code is automatically compiled by Maven.

Reference

For full details of how to configure the Java-to-WSDL plug-in, see the Maven cxf-codegen-plugin reference page.

4.3. INSTANTIATE THE WS CLIENT PROXY

Overview

The WS client proxy is the most important kind of object in a WS client, because it provides a simple way of invoking operations on a remote Web service. The proxy enables you to access a Web service by invoking methods locally on a Java interface. The methods invoked on the proxy object are then translated into remote procedure calls on the Web service.

You can instantiate a WS client proxy straightforwardly using the **jaxws:client** element.

Define the WS client in XML

The following Spring XML fragment shows how to instantiate a client proxy bean using the **jaxws:client** element.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jaxws="http://cxf.apache.org/jaxws"
      xmlns:soap="http://cxf.apache.org/bindings/soap"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/bindings/soap
http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
```

```
  <jaxws:client
    id="customerServiceProxy"
    address="http://localhost:8181/cxf/Customers"

serviceClass="com.fusesource.demo.wsdl.customerservice.CustomerService"
    />

  <bean id="customerServiceClient"
      class="com.fusesource.customer.client.ClientInvoker"
      init-method="init" destroy-method="destroy">
       <property name="customerService" ref="customerServiceProxy"/>
  </bean>

</beans>
```

The jaxws:client element

The **jaxws:client** element creates a client proxy *dynamically* (that is, there is no dedicated class that represents a proxy implementation in the Java stub code). The following attributes are used to define the proxy:

id

The ID that you specify here is entered in the bean registry and can be used to reference the proxy instance from other beans.

address

The full address of the remote Web service that this proxy connects to.

serviceClass

The fully-qualified class name of the Web service's SEI (you invoke methods on the proxy through the SEI).

Injecting with the proxy reference

To access the proxy instance, simply inject the proxy into one or more other beans defined in XML. Given that the proxy ID has the value, **customerServiceProxy**, you can inject it into a bean property using the Spring **property** element, as follows:

```
<bean ...>
  <property name="customerService" ref="customerServiceProxy"/>
</bean>
```

The bean class that is being injected must have a corresponding **setCustomerService** setter method —for example:

```
// Java
...
public class ClientInvoker implements Runnable {
  ...
  public void setCustomerService(CustomerService customerService) {
    this.customerService = customerService;
  }
```

```
}
```

4.4. INVOKE WS OPERATIONS

Proxy interface is SEI interface

The proxy implements the SEI. Hence, to make remote procedure calls on the Web service, simply invoke the SEI methods on the proxy instance.

Invoking the lookupCustomer operation

For example, the **`CustomerService`** SEI exposes the **`lookupCustomer`** method, which takes a customer ID as its argument and returns a **`Customer`** data object. Using the proxy instance, **`customerService`**, you can invoke the **`lookupCustomer`** operation as follows:

```
// Java
com.fusesource.demo.customer.Customer response
 = customerService.lookupCustomer("1234");

log.info("Got back " + response.getFirstName() + " "
    + response.getLastName()
    + ", ph:" + response.getPhoneNumber() );
```

The ClientInvoker class

In the **`cxf-webinars/customer-ws-client`** project, there is a **`ClientInvoker`** class (located in **`src/main/java/com/fusesource/customer/client`**), which defines a continuous loop that invokes the **`lookupCustomer`** operation.

When you are experimenting with the demonstration code in the latter chapters of this guide, you might need to modify the **`ClientInvoker`** class, possibly adding operation invocations.

4.5. DEPLOY TO AN OSGI CONTAINER

Overview

One of the options for deploying the WS client is to package it as an OSGi bundle and deploy it into an OSGi container such as Red Hat JBoss Fuse. Some of the advantages of an OSGi deployment include:

- Bundles are a relatively lightweight deployment option (because dependencies can be shared between deployed bundles).
- OSGi provides sophisticated dependency management, ensuring that only version-consistent dependencies are added to the bundle's classpath.

Using the Maven bundle plug-in

The Maven bundle plug-in is used to package your project as an OSGi bundle, in preparation for deployment into the OSGi container. There are two essential modifications to make to your project's **`pom.xml`** file:

1. Change the packaging type to **bundle** (by editing the value of the **project/packaging** element in the POM).

2. Add the Maven bundle plug-in to your POM file and configure it as appropriate.

Configuring the Maven bundle plug-in is quite a technical task (although the default settings are often adequate). For full details of how to customize the plug-in configuration, consult *Deploying into the OSGi Container* and *Managing OSGi Dependencies*.

Sample bundle plug-in configuration

The following POM fragment shows a sample configuration of the Maven bundle plug-in, which is appropriate for the current example.

```
<?xml version="1.0"?>
<project  ...>
  ...
  <groupId>org.fusesource.sparks.fuse-webinars.cxf-webinars</groupId>
  <artifactId>customer-ws-osgi-bundle</artifactId>
  <name>customer-ws-osgi-bundle</name>
  <url>http://www.fusesource.com</url>
 <packaging>bundle</packaging>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>${maven-bundle-plugin.version}</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Export-Package>
              !com.fusesource.customer.client,
              !com.fusesource.demo.customer,
              !com.fusesource.demo.wsdl.customerservice
            </Export-Package>
            <Import-Package>
              META-INF.cxf,
              *
            </Import-Package>
            <DynamicImport-Package>
              org.apache.cxf.*,
              org.springframework.beans.*
            </DynamicImport-Package>
          </instructions>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
</project>
```

Dynamic imports

The Java packages from Apache CXF and the Spring API are imported using dynamic imports (specified using the **DynamicImport-Package** element). This is a pragmatic way of dealing with the fact that Spring XML files are not terribly well integrated with the Maven bundle plug-in. At build time, the Maven bundle plug-in is *not* able to figure out which Java classes are required by the Spring XML code. By listing wildcarded package names in the **DynamicImport-Package** element, however, you allow the OSGi container to figure out which Java classes are needed by the Spring XML code at *run* time.

NOTE

In general, using **DynamicImport-Package** headers is not recommended in OSGi, because it short-circuits OSGi version checking. Normally, what should happen is that the Maven bundle plug-in lists the Java packages used at *build* time, along with their versions, in the **Import-Package** header. At *deploy* time, the OSGi container then checks that the available Java packages are compatible with the build time versions listed in the **Import-Package** header. With dynamic imports, this version checking cannot be performed.

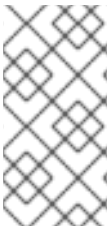
Build and deploy the client bundle

After you have configured the POM file, you can build the Maven project and install it in your local repository by entering the following command:

```
mvn install
```

To deploy the client bundle, enter the following command at the containers command console:

```
karaf@root> install -s mvn:org.fusesource.sparks.fuse-webinars.cxf-
webinars/customer-ws-client
```



NOTE

If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **org.ops4j.pax.url.mvn.localRepository** property in the ***EsbInstallDir*/etc/org.ops4j.pax.url.mvn.cfg** file, before you can use the **mvn:** scheme to access Maven artifacts.

Check that the client is running

Assuming that you have already deployed the corresponding Web service into the OSGi container, you can verify that the client is successfully invoking WSDL operations by checking the log, as follows:

```
karaf@root> log:display -n 10
```

The client invokes an operation on the Web service once every second.

CHAPTER 5. POJO-BASED ROUTE

5.1. PROCESSING MESSAGES IN POJO FORMAT

Overview

By default, the Camel CXF component marshals incoming Web service requests into the POJO data form, where the *In* message body is encoded as a list of Java objects (one for each operation parameter). The POJO data format has advantages and disadvantages, as follows:

- The big advantage of the POJO data format is that the operation parameters are encoded using the JAX-B standard, which makes them easy to manipulate in Java.
- The downside of the POJO data format, on the other hand, is that it requires that the WSDL metadata is converted to Java in advance (as defined by the JAX-WS and JAX-B mappings) and compiled into your application. This means that a POJO-based route is not very dynamic.

Demonstration location

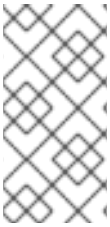
The code presented in this chapter is taken from the following demonstration:

```
DemoDir/src/fuse-webinars/cxf-webinars/customer-ws-camel-cxf-pojo
```

For details of how to download and install the demonstration code, see Chapter 1, *Demonstration Code for Camel/CXF*

Camel CXF component

The Camel CXF component is an Apache CXF component that integrates Web services with routes. You can use it either to instantiate *consumer endpoints* (at the start of a route), which behave like Web service instances, or to instantiate *producer endpoints* (at any other points in the route), which behave like WS clients.



NOTE

Camel CXF endpoints—which are instantiated using the **`cxf:cxfEndpoint`** XML element and are implemented by the Apache Camel project—are not to be confused with the Apache CXF JAX-WS endpoints—which are instantiated using the **`jaxws:endpoint`** XML element and are implemented by the Apache CXF project.

POJO data format

POJO data format is the *default* data format used by the Camel CXF component and it has the following characteristics:

- JAX-WS and JAX-B stub code (as generated from the WSDL contract) *must* be provided.
- The SOAP body is marshalled into a list of Java objects.
  - One Java object for each part or parameter of the corresponding WSDL operation.
  - The type of the message body is **`org.apache.cxf.message.MessageContentsList`**.

- The SOAP headers are converted into headers in the exchange's *In* message.

Implementing and building a POJO route

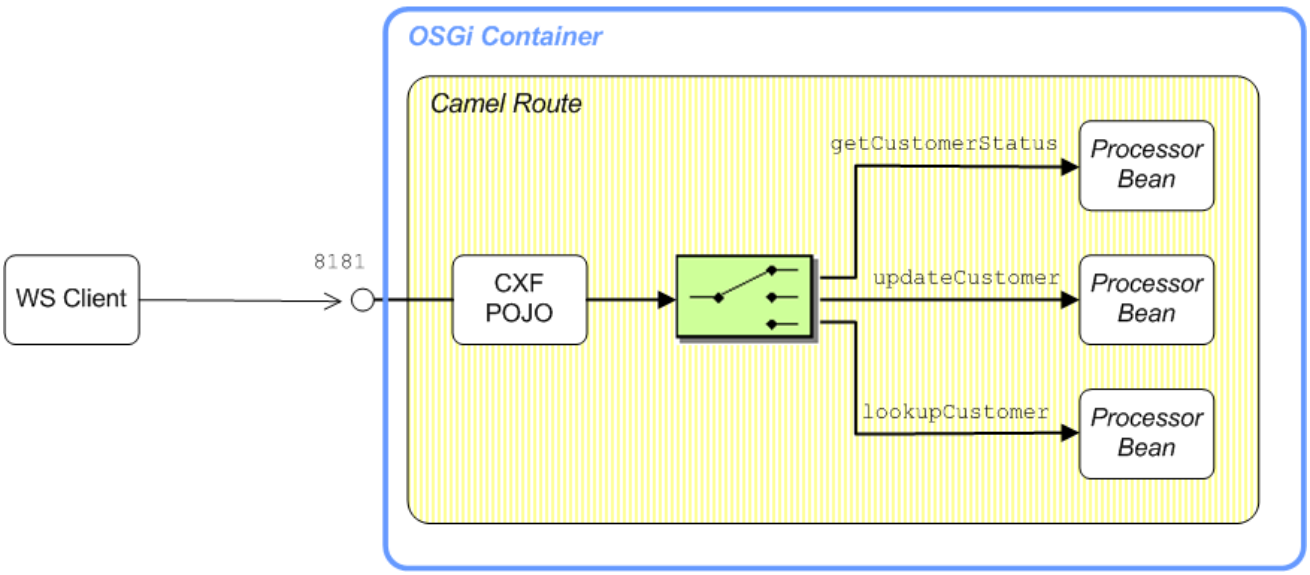
To implement and build the demonstration POJO-based route, starting from scratch, you would perform the following steps:

1. Obtain a copy of the WSDL contract that is to be integrated into the route.
2. Generate the Java stub code from the WSDL contract using a WSDL-to-Java converter. This gives you the SEI, **`CustomerService`**, and its related classes, such as **`Customer`**.
3. Instantiate the Camel CXF endpoint in Spring, using the **`cxf:cxfEndpoint`** element.
4. Implement the route in XML, where you can use the content-based router to sort requests by operation name.
5. Implement the operation processor beans, which are responsible for processing each operation. When implementing these beans, the message contents must be accessed in POJO data format.

Sample POJO route

Figure 5.1, “Sample POJO Route” shows an outline of the route that is used to process the operations of the **`CustomerService`** Web service using the POJO data format. After sorting the request messages by operation name, an operation-specific processor bean reads the incoming request parameters and then generates a response in the POJO data format.

Figure 5.1. Sample POJO Route



5.2. WSDL-TO-JAVA MAVEN PLUG-IN

Overview

To generate Java stub code from the WSDL contract, you can use either the **`ws2java`** command-line utility or the **`cxf-codegen-plugin`** Maven plug-in. When using Maven, the plug-in approach is ideal: after you paste the requisite plug-in configuration into your POM file, the WSDL-to-Java code generation step is integrated into your build.

Configure the WSDL-to-Java Maven plug-in

Configuring the WSDL-to-Java Maven plug-in is relatively easy, because most of the default configuration settings can be left as they are. After copying and pasting the sample **`plugin`** element into your project's POM file, there are just a few basic settings that need to be customized, as follows:

- *CXF version*—make sure that the plug-in's dependencies are using the latest version of Apache CXF.
- *WSDL file location*—specify the WSDL file location in the **`configuration/wsdlOptions/wsdlOption/wsdl`** element.
- *Location of output*—specify the root directory of the generated Java source files in the **`configuration/sourceRoot`** element.

For example, the following POM fragment shows how to configure the **`cxf-codegen-plugin`** plug-in to generate Java stub code from the **`CustomerService.wsdl`** WSDL file:

```
<project ...>
  ...
  <properties>
    <cxf.version>2.6.0.redhat-60024</cxf.version>
  </properties>

  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-codegen-plugin</artifactId>
        <version>${cxf.version}</version>
        <executions>
          <execution>
            <id>generate-sources</id>
            <phase>generate-sources</phase>
            <configuration>
              <sourceRoot>${basedir}/target/generated-
sources/jaxws</sourceRoot>
              <wsdlOptions>
                <wsdlOption>

<wsdl>${basedir}/../src/main/resources/wsdl/CustomerService.wsdl</wsdl>
                </wsdlOption>
              </wsdlOptions>
            </configuration>
            <goals>
              <goal>wsdl2java</goal>
            </goals>
          </execution>
        </executions>
      </plugin>

    </plugins>
```

```
  </build>

</project>
```

Generated Java source code

With the sample configuration shown here, the generated Java source code is written under the **`target/generated-sources/jaxws`** directory. Note that the route is dependent on this generated stub code—for example, when processing the POJO parameters, the parameter processor uses the **`Customer`** data type from the stub code.

Add generated code to IDE

If you are using an IDE such as Eclipse or Intellij's IDEA, you need to make sure that the IDE is aware of the generated Java code. For example, in Eclipse it is necessary to add the **`target/generated-sources/jaxws`** directory to the project as a source code directory.

Compiling the generated code

You must ensure that the generated Java code is compiled and added to the deployment package. By convention, Maven automatically compiles any source files that it finds under the following directory:

```
BaseDir/target/generated-sources/
```

Hence, if you configure the output directory as shown in the preceding POM fragment, the generated code is automatically compiled by Maven.

Reference

For full details of how to configure the Java-to-WSDL plug-in, see the Maven cxf-codegen-plugin reference page.

5.3. INSTANTIATE THE WS ENDPOINT

Overview

In Apache Camel, the Camel CXF component is the key to integrating routes with Web services. You can use the Camel CXF component to create a CXF endpoint, which can be used in either of the following ways:

- *Consumer*—(at the start of a route) represents a Web service instance, which integrates with the route. The type of payload injected into the route depends on the value of the endpoint's **`dataFormat`** option.
- *Producer*—(at other points in the route) represents a WS client proxy, which converts the current exchange object into an operation invocation on a remote Web service. The format of the current exchange must match the endpoint's **`dataFormat`** setting.

In the current demonstration, we are interested in creating a Camel CXF consumer endpoint, with the **`dataFormat`** option set to POJO.

Maven dependency

The Camel CXF component requires you to add a dependency on the **camel-cxf** component in your Maven POM. For example, the **pom.xml** file from the **customer-ws-camel-cxf-pojo** demonstration project includes the following dependency:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cxf</artifactId>
  <version>${camel-version}</version>
</dependency>
```

The cxf:bean: URI syntax

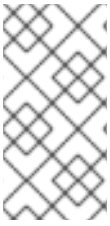
The **cxf:bean:** URI is used to bind an Apache CXF endpoint to a route and has the following general syntax:

```
cxf:bean:CxfEndpointID[?Options]
```

Where ***CxfEndpointID*** is the ID of a bean created using the **cxf:cxfEndpoint** element, which configures the details of the WS endpoint. You can append options to this URI (where the options are described in detail in chapter "CXF" in "EIP Component Reference"). If you do not specify any additional options, *the endpoint uses the POJO data format by default*.

For example, to start a route with a Apache CXF endpoint that is configured by the bean with ID, **customer-ws**, define the route as follows:

```
<route>
    <from uri="cxf:bean:customer-ws"/>
    ...
</route>
```



NOTE

There is an alternative URI syntax, **cxf://*WsAddress*[?*Options*]**, which enables you to specify *all* of the WS endpoint details in the URI (so there is no need to reference a bean instance). This typically results in a long and cumbersome URI, but is useful in some cases.

The cxf:cxfEndpoint element

The **cxf:cxfEndpoint** element is used to define a WS endpoint that binds either to the start (consumer endpoint) or the end (producer endpoint) of a route. For example, to define the **customer-ws** WS endpoint referenced in the preceding route, you would define a **cxf:cxfEndpoint** element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...
   xmlns:cxf="http://camel.apache.org/schema/cxf"  ...>
    ...
    <cxf:cxfEndpoint id="customer-ws"
        address="/Customer"
        endpointName="c:SOAPOverHTTP"
        serviceName="c:CustomerService"
```

```
serviceClass="com.fusesource.demo.wsdl.customerservice.CustomerService"
        xmlns:c="http://demo.fusesource.com/wsdl/CustomerService/"/>
    ...
</beans>
```

IMPORTANT

Remember that the **cxf:cxfEndpoint** element and the **jaxws:endpoint** element use *different* XML schemas (although the syntax looks superficially similar). These elements bind a WS endpoint in different ways: the **cxf:cxfEndpoint** element instantiates and binds a WS endpoint to an Apache Camel route, whereas the **jaxws:endpoint** element instantiates and binds a WS endpoint to a Java class using the JAX-WS mapping.

Address for the Jetty container

Apache CXF deploys the WS endpoint into a Jetty servlet container instance and the **address** attribute of **cxf:cxfEndpoint** is therefore used to configure the addressing information for the endpoint in the Jetty container.

Apache CXF supports the notion of a *default servlet container* instance. The way the default servlet container is initialized and configured depends on the particular mode of deployment that you choose. For example the Red Hat JBoss Fuse container and Web containers (such as Tomcat) provide a default servlet container.

There are two different syntaxes you can use for the endpoint address, where the syntax that you use effectively determines whether or not the endpoint is deployed into the default servlet container, as follows:

- *Address syntax for default servlet container*—to use the default servlet container, specify only the servlet context for this endpoint. Do *not* specify the protocol, host, and IP port in the address. For example, to deploy the endpoint to the **/Customers** servlet context in the default servlet container:

  ```
  address="/Customers"
  ```

- *Address syntax for custom servlet container*—to instantiate a custom Jetty container for this endpoint, specify a complete HTTP URL, including the host and IP port (the value of the IP port effectively identifies the target Jetty container). Typically, for a Jetty container, you specify the host as **0.0.0.0**, which is interpreted as a wildcard that matches every IP network interface on the local machine (that is, if deployed on a multi-homed host, Jetty opens a listening port on every network card). For example, to deploy the endpoint to the custom Jetty container listening on IP port, **8083**:

  ```
  address="http://0.0.0.0:8083/Customers"
  ```

  NOTE

  If you want to configure a secure endpoint (secured by SSL), you would specify the **https:** scheme in the address.

Referencing the SEI

The **serviceClass** attribute of the **cxf:cxfEndpoint** element references the SEI of the Web service, which in this case is the **CustomerService** interface.

5.4. SORT MESSAGES BY OPERATION NAME

The operationName header

When the WS endpoint parses an incoming operation invocation in POJO mode, it automatically sets the **operationName** header to the name of the invoked operation. You can then use this header to sort messages by operation name.

Sorting by operation name

For example, the **customer-ws-camel-cxf-pojo** demonstration defines the following route, which uses the content-based router pattern to sort incoming messages, based on the operation name. The **when** predicates check the value of the **operationName** header using simple language expressions, sorting messages into invocations on the **updateCustomer** operation, the **lookupCustomer** operation, or the **getCustomerStatus** operation.

```
<beans ...>
    ...
    <camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="cxf:bean:customer-ws"/>
            <choice>
                <when>
                    <simple>${in.header.operationName} ==
'updateCustomer'</simple>
                    <to uri="updateCustomer"/>
                </when>
                <when>
                    <simple>${in.header.operationName} ==
'lookupCustomer'</simple>
                    <to uri="lookupCustomer"/>
                </when>
                <when>
                    <simple>${in.header.operationName} ==
'getCustomerStatus'</simple>
                    <to uri="getCustomerStatus"/>
                </when>
            </choice>
        </route>
    </camelContext>

    <bean id="updateCustomer"

class="com.fusesource.customerwscamelcxfpojo.UpdateCustomerProcessor"/>

    <bean id="getCustomerStatus"

class="com.fusesource.customerwscamelcxfpojo.GetCustomerStatusProcessor"/>

    <bean id="lookupCustomer"
```

```
class="com.fusesource.customerwscamelcxfpojo.LookupCustomerProcessor"/>

</beans>
```

Beans as endpoints

Note how the preceding route uses a convenient shortcut to divert each branch of the **choice** DSL to a different processor bean. The DSL for sending exchanges to producer endpoints (for example, **<to uri="*Destination*"/>**) is integrated with the bean registry: if the *Destination* does not resolve to an endpoint or a component, the *Destination* is used as a bean ID to look up the bean registry. In this example, the exchange is routed to processor beans (which implement the **org.apache.camel.Processor** interface).

5.5. PROCESS OPERATION PARAMETERS

Overview

The most important characteristic of using Camel CXF in POJO mode is that the exchange's message body contains a list of Java objects, representing the parameters of the WSDL operation. The types of the Java objects are defined by the standard JAX-B mapping and the implementations of these parameter types are provided by the Java stub code.

Contents of request message body

In POJO mode, the body of the request message is an **org.apache.cxf.message.MessageContentsList** object. You can also obtain the message body as an **Object[]** array (where type conversion is automatic).

When the body is obtained as an **Object[]** array, the array contains the list of all the operation's IN, INOUT, and OUT parameters *in exactly the same order as defined in the WSDL contract* (and in the same order as the corresponding operation signature of the SEI). The parameter mode affects the content as follows:

IN

Contains a parameter value from the client.

INOUT

Contains a **Holder** object containing a parameter value from the client.

OUT

Contains an *empty* **Holder** object, which is a placeholder for the response.

NOTE

Unlike OUT parameters, there is no placeholder in the request's **Object[]** array to represent a return value.

Contents of response message body

In POJO mode, the body of the response message can be either an **`org.apache.cxf.message.MessageContentsList`** object or an **`Object[]`** array.

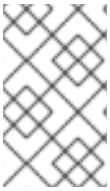
When setting the response body as an **`Object[]`** array, the array should contain *only* the operation's INOUT and OUT parameters in the same order as defined in the WSDL contract, omitting the IN parameters. The parameter mode affects the content as follows:

`INOUT`

Contains a **`Holder`** object, which you must set to a response value. The **`Holder`** object used here *must be exactly the* ***`Holder`*** *object for the corresponding parameter that was extracted from the request* ***`Object[]`*** *array*. Creating and inserting a new **`Holder`** object into the **`Object[]`** array does *not* work.

`OUT`

Contains a **`Holder`** object, which you must initialize with a response value. The **`Holder`** object used here *must be exactly the* ***`Holder`*** *object for the corresponding parameter that was extracted from the request* ***`Object[]`*** *array*. Creating and inserting a new **`Holder`** object into the **`Object[]`** array does *not* work.



NOTE

If you defined the Web service interface using the Java-first approach, note that the return value (if any) must be set as the *first* element in the response's **`Object[]`** array. The return type is set as a plain object: it does *not* use a **`Holder`** object.

Example: getCustomerStatus operation

For example, the **`getCustomerStatus`** operation takes three parameters: IN, OUT, and OUT, respectively. The corresponding method signature in the SEI is, as follows:

```
// Java
public void getCustomerStatus(
    @WebParam(name = "customerId", targetNamespace = "")
    java.lang.String customerId,

    @WebParam(mode = WebParam.Mode.OUT, name = "status", targetNamespace =
"")
    javax.xml.ws.Holder<java.lang.String> status,

    @WebParam(mode = WebParam.Mode.OUT, name = "statusMessage",
targetNamespace = "")
    javax.xml.ws.Holder<java.lang.String> statusMessage
);
```

Example: request and response bodies

For the **`getCustomerStatus`** operation, the bodies of the request message and the response message have the following contents:

- *Request message*—as an **`Object[]`** array type, the contents are: **`{ String customerId, Holder<String> status, Holder<String> statusMessage }`**.

- *Response message*—as an **`Object[]`** array type, the contents are: **`{Holder<String> status, Holder<String> statusMessage }`**

Example: processing getCustomerStatus

The **`GetCustomerStatusProcessor`** class is responsible for processing incoming **`getCustomerStatus`** invocations. The following sample implementation for POJO mode shows how to read the request parameters from the *In* message body and then set the response parameters in the *Out* message body.

```
// Java
package com.fusesource.customerwscamelcxfpojo;

import javax.xml.ws.Holder;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


public class GetCustomerStatusProcessor implements Processor {
    public static final Logger log =
LoggerFactory.getLogger(GetCustomerStatusProcessor.class);

    public void process(Exchange exchng) throws Exception {
        Object[] args =  exchng.getIn().getBody(Object[].class);

        String id = (String) args[0];
        Holder<String> status = (Holder<String>) args[1];
        Holder<String> statusMsg = (Holder<String>) args[2];

        log.debug("Getting status for customer '" + id + "'");

        // This is where you'd actually do the work! Setting
        // the holder values to constants for the sake of brevity.
        //
        status.value = "Offline";
        statusMsg.value = "Going to sleep now!";

        exchng.getOut().setBody(new Object[] {status , statusMsg});
    }

}
```

5.6. DEPLOY TO OSGI

Overview

One of the options for deploying the POJO-based route is to package it as an OSGi bundle and deploy it into an OSGi container such as Red Hat JBoss Fuse. Some of the advantages of an OSGi deployment include:

- Bundles are a relatively lightweight deployment option (because dependencies can be shared between deployed bundles).

- OSGi provides sophisticated dependency management, ensuring that only version-consistent dependencies are added to the bundle's classpath.

Using the Maven bundle plug-in

The Maven bundle plug-in is used to package your project as an OSGi bundle, in preparation for deployment into the OSGi container. There are two essential modifications to make to your project's **`pom.xml`** file:

1. Change the packaging type to **`bundle`** (by editing the value of the **`project/packaging`** element in the POM).
2. Add the Maven bundle plug-in to your POM file and configure it as appropriate.

Configuring the Maven bundle plug-in is quite a technical task (although the default settings are often adequate). For full details of how to customize the plug-in configuration, consult *Deploying into the OSGi Container* and *Managing OSGi Dependencies*.

Sample bundle plug-in configuration

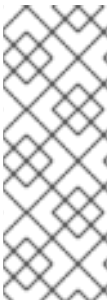
The following POM fragment shows a sample configuration of the Maven bundle plug-in, which is appropriate for the current example.

```
<?xml version="1.0"?>
<project  ...>
  ...
  <groupId>org.fusesource.sparks.fuse-webinars.cxf-webinars</groupId>
  <artifactId>customer-ws-camel-cxf-pojo</artifactId>

  <name>customer-ws-camel-cxf-pojo</name>
  <url>http://www.fusesource.com</url>
 <packaging>bundle</packaging>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
         <instructions>
            <Import-Package>
              META-INF.cxf,
              META-INF.cxf.osgi,
              *
            </Import-Package>
            <DynamicImport-Package>
              org.apache.cxf.*,
              org.springframework.beans.*
            </DynamicImport-Package>
          </instructions>
        </configuration>
      </plugin>
      ...
```

```
    </plugins>
  </build>
</project>
```

Dynamic imports

The Java packages from Apache CXF and the Spring API are imported using dynamic imports (specified using the **DynamicImport-Package** element). This is a pragmatic way of dealing with the fact that Spring XML files are not terribly well integrated with the Maven bundle plug-in. At build time, the Maven bundle plug-in is *not* able to figure out which Java classes are required by the Spring XML code. By listing wildcarded package names in the **DynamicImport-Package** element, however, you allow the OSGi container to figure out which Java classes are needed by the Spring XML code at *run* time.



NOTE

In general, using **DynamicImport-Package** headers is not recommended in OSGi, because it short-circuits OSGi version checking. Normally, what should happen is that the Maven bundle plug-in lists the Java packages used at *build* time, along with their versions, in the **Import-Package** header. At *deploy* time, the OSGi container then checks that the available Java packages are compatible with the build time versions listed in the **Import-Package** header. With dynamic imports, this version checking cannot be performed.

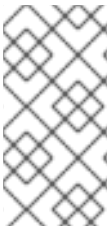
Build and deploy the POJO route bundle

After you have configured the POM file, you can build the Maven project and install it in your local repository by entering the following command:

```
mvn install
```

To deploy the route bundle, enter the following command at the Apache ServiceMix console:

```
karaf@root> install -s mvn:org.fusesource.sparks.fuse-webinars.cxf-
webinars/customer-ws-camel-cxf-pojo
```



NOTE

If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **org.ops4j.pax.url.mvn.localRepository** property in the ***EsbInstallDir*/etc/org.ops4j.pax.url.mvn.cfg** file, before you can use the **mvn:** scheme to access Maven artifacts.

CHAPTER 6. PAYLOAD-BASED ROUTE

6.1. PROCESSING MESSAGES IN PAYLOAD FORMAT

Overview

Select the PAYLOAD format, if you want to access the SOAP message body in XML format, encoded as a DOM object (that is, of **`org.w3c.dom.Node`** type). One of the advantages of the PAYLOAD format is that no JAX-WS and JAX-B stub code is required, which allows your application to be dynamic, potentially handling many different WSDL interfaces.

Having a message body in XML format enables you to parse the request using XML languages such as XPath and to generate responses using templating languages, such as Velocity.

NOTE

The DOM format is not the optimal type to use for *large* XML message bodies. For large messages, consider using the techniques described in Chapter 7, *Provider-Based Route*.

Demonstration location

The code presented in this chapter is taken from the following demonstration:

> *DemoDir*`/src/fuse-webinars/cxf-webinars/`**`customer-ws-camel-cxf-payload`**

For details of how to download and install the demonstration code, see Chapter 1, *Demonstration Code for Camel/CXF*

Camel CXF component

The Camel CXF component is an Apache CXF component that integrates Web services with routes. You can use it either to instantiate *consumer endpoints* (at the start of a route), which behave like Web service instances, or to instantiate *producer endpoints* (at any other points in the route), which behave like WS clients.

NOTE

Came CXF endpoints—which are instantiated using the **`cxf:cxfEndpoint`** XML element and are implemented by the Apache Camel project—are not to be confused with the Apache CXF JAX-WS endpoints—which are instantiated using the **`jaxws:endpoint`** XML element and are implemented by the Apache CXF project.

PAYLOAD data format

The PAYLOAD data format is selected by setting the **`dataFormat=PAYLOAD`** option on a Camel CXF endpoint URI and it has the following characteristics:

- Enables you to access the message body as a DOM object (XML payload).
- No JAX-WS or JAX-B stub code required.
- The SOAP body is marshalled as follows:

- The message body is effectively an XML payload of **`org.w3c.dom.Node`** type (wrapped in a **`CxfPayload`** object).
- The type of the message body is **`org.apache.camel.component.cxf.CxfPayload`**.

- The SOAP headers are converted into headers in the exchange's *In* message, of **`org.apache.cxf.binding.soap.SoapHeader`** type.

Implementing and building a PAYLOAD route

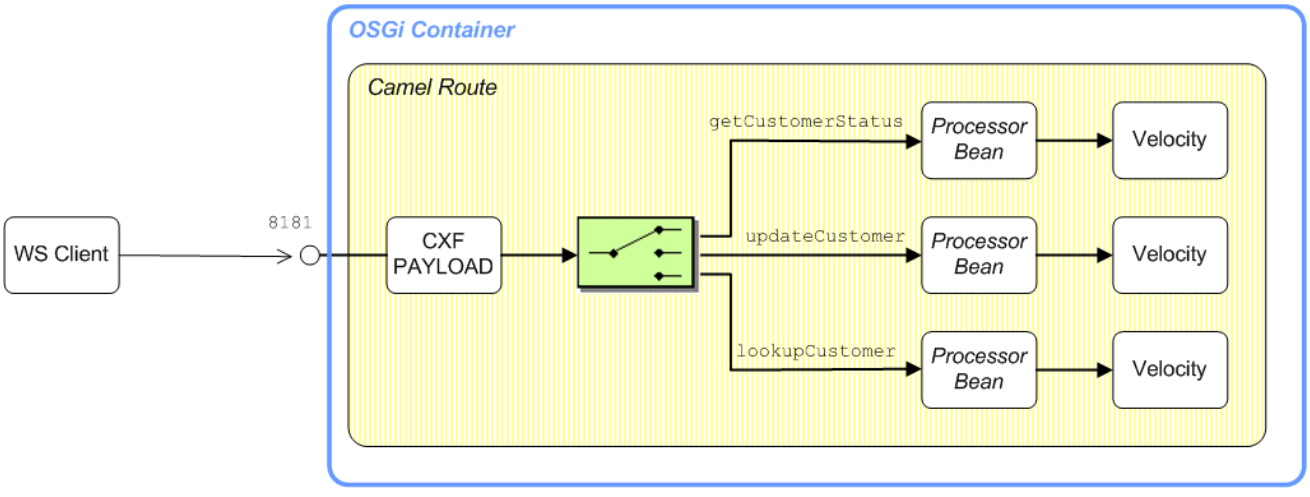
To implement and build the demonstration PAYLOAD-based route, starting from scratch, you would perform the following steps:

1. Instantiate the Camel CXF endpoint in Spring, using the **`cxf:cxfEndpoint`** element.
2. Implement the route in XML, where you can use the content-based router to sort requests by operation name.
3. For each operation, define a processor bean to process the request.
4. Define velocity templates for generating the reponse messages.

Sample PAYLOAD route

Figure 6.1, “Sample PAYLOAD Route” shows an outline of the route that is used to process the operations of the **`CustomerService`** Web service using the PAYLOAD data format. After sorting the request messages by operation name, an operation-specific processor bean reads the incoming request parameters. Finally, the response messages are generated using Velocity templates.

Figure 6.1. Sample PAYLOAD Route



6.2. INSTANTIATE THE WS ENDPOINT

Overview

In Apache Camel, the CXF component is the key to integrating routes with Web services. You can use the CXF component to create two different kinds of endpoint:

- *Consumer endpoint*—(at the start of a route) represents a Web service instance, which integrates with the route. The type of payload injected into the route depends on the value of the endpoint's **dataFormat** option.
- *Producer endpoint*—represents a special kind of WS client proxy, which converts the current exchange object into an operation invocation on a remote Web service. The format of the current exchange must match the endpoint's **dataFormat** setting.

The cxf:bean: URI syntax

The **cxf:bean:** URI is used to bind an Apache CXF endpoint to a route and has the following general syntax:

```
cxf:bean:CxfEndpointID[?Options]
```

Where ***CxfEndpointID*** is the ID of a bean created using the **cxf:cxfEndpoint** element, which configures the details of the WS endpoint. You can append options to this URI (where the options are described in detail in chapter "CXF" in "EIP Component Reference"). To enable payload mode, you must set the URI option, **dataFormat=PAYLOAD**.

For example, to start a route with an endpoint in PAYLOAD mode, where the endpoint is configured by the **customer-ws** bean, define the route as follows:

```
<route>
    <from uri="cxf:bean:customer-ws?dataFormat=PAYLOAD"/>
    ...
</route>
```

The cxf:cxfEndpoint element

The **cxf:cxfEndpoint** element is used to define a WS endpoint that binds either to the start (consumer endpoint) or the end (producer endpoint) of a route. For example, to define the **customer-ws** WS endpoint in PAYLOAD mode, you define a **cxf:cxfEndpoint** element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    ...
    <cxf:cxfEndpoint id="customer-ws"
        address="/Customer"
        endpointName="c:SOAPOverHTTP"
        serviceName="c:CustomerService"
        wsdlURL="wsdl/CustomerService.wsdl"
        xmlns:c="http://demo.fusesource.com/wsdl/CustomerService/"/>
    ...
</beans>
```

NOTE

In the case of PAYLOAD mode, you do *not* need to reference the SEI and you *must* specify the WSDL location instead. In fact, in PAYLOAD mode, you do not require any Java stub code at all.

Address for the Jetty container

Apache CXF deploys the WS endpoint into a Jetty servlet container instance and the **address** attribute of **cxf:cxfEndpoint** is therefore used to configure the addressing information for the endpoint in the Jetty container.

Apache CXF supports the notion of a *default servlet container* instance. The way the default servlet container is initialized and configured depends on the particular mode of deployment that you choose. For example the OSGi container and Web containers (such as Tomcat) provide a default servlet container.

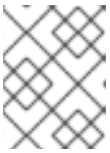
There are two different syntaxes you can use for the endpoint address, where the syntax that you use effectively determines whether or not the endpoint is deployed into the default servlet container, as follows:

- *Address syntax for default servlet container*—to use the default servlet container, specify only the servlet context for this endpoint. Do *not* specify the protocol, host, and IP port in the address. For example, to deploy the endpoint to the **/Customers** servlet context in the default servlet container:

  ```
  address="/Customers"
  ```

- *Address syntax for custom servlet container*—to instantiate a custom Jetty container for this endpoint, specify a complete HTTP URL, including the host and IP port (the value of the IP port effectively identifies the target Jetty container). Typically, for a Jetty container, you specify the host as **0.0.0.0**, which is interpreted as a wildcard that matches every IP network interface on the local machine (that is, if deployed on a multi-homed host, Jetty opens a listening port on every network card). For example, to deploy the endpoint to the custom Jetty container listening on IP port, **8083**:

  ```
  address="http://0.0.0.0:8083/Customers"
  ```



NOTE

If you want to configure a secure endpoint (secured by SSL), you would specify the **https:** scheme in the address.

Specifying the WSDL location

The **wsdlURL** attribute of the **cxf:cxfEndpoint** element is used to specify the location of the WSDL contract for this endpoint. The WSDL contract is used exclusively as the source of metadata for this endpoint: there is need to specify an SEI in PAYLOAD mode.

6.3. SORT MESSAGES BY OPERATION NAME

The operationName header

When the WS endpoint parses an incoming operation invocation in PAYLOAD mode, it automatically sets the **operationName** header to the name of the invoked operation. You can then use this header to sort messages by operation name.

Sorting by operation name

For example, the **customer-ws-camel-cxf-payload** demonstration defines the following route, which uses the content-based router pattern to sort incoming messages, based on the operation name. The **when** predicates check the value of the **operationName** header using simple language expressions, sorting messages into invocations on the **updateCustomer** operation, the **lookupCustomer** operation, or the **getCustomerStatus** operation.

```
<beans ...>
    ...
    <camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="cxf:bean:customer-ws?dataFormat=PAYLOAD"/>
            <choice>
                <when>
                    <simple>${in.header.operationName} ==
'updateCustomer'</simple>
                    ...
                </when>
                <when>
                    <simple>${in.header.operationName} ==
'lookupCustomer'</simple>
                    ...
                </when>
                <when>
                    <simple>${in.header.operationName} ==
'getCustomerStatus'</simple>
                    ...
                </when>
            </choice>
        </route>
    </camelContext>

</beans>
```

6.4. SOAP/HTTP-TO-JMS BRIDGE USE CASE

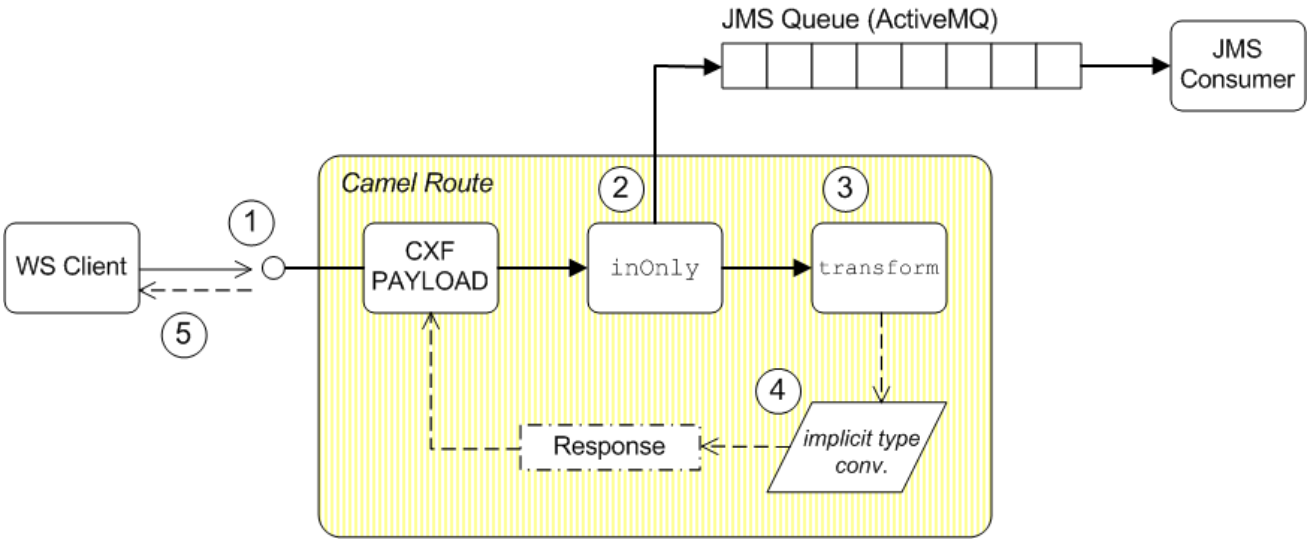
Overview

In this section, we consider a SOAP/HTTP-to-JMS bridge use case: that is, you want to create a route that transforms a synchronous operation invocation (over SOAP/HTTP) into an asynchronous message delivery (by pushing the message onto a JMS queue). In this way, it becomes possible to process the incoming operation invocations at a later time, by pulling messages off the JMS queue.

Of course, an alternative solution would be to modify the WSDL contract directly to declare the operation as *OneWay*, thus making the operation asynchronous. Unfortunately, it is often impractical to modify existing WSDL contracts in the real world, because this can have an impact on third-party applications.

Figure 6.2, “SOAP/HTTP-to-JMS Bridge” shows the general outline of a bridge that can transform synchronous SOAP/HTTP invocations into asynchronous JMS message deliveries.

Figure 6.2. SOAP/HTTP-to-JMS Bridge



Transforming RPC operations to One Way

As shown in Figure 6.2, “SOAP/HTTP-to-JMS Bridge”, the route for transforming synchronous SOAP/HTTP to asynchronous JMS works as follows:

1. The WS client invokes a synchronous operation on the Camel CXF endpoint at the start of the route. The Camel CXF endpoint then creates an initial *InOut* exchange at the start of the route, where the body of the exchange message contains a payload in XML format.
2. The **`inOnly`** DSL command pushes a copy of the XML payload onto a JMS queue, so that it can be processed offline at some later time.
3. The **`transform`** DSL command constructs an immediate response to send back to the client, where the response has the form of an XML string.
4. The Camel CXF component supports implicit type conversion of the XML string to payload format.
5. The response is sent back to the WS client, thus completing the synchronous operation invocation.

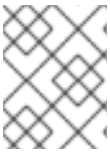
Evidently, this transformation can only work, if the original operation invocation has no return value. Otherwise, it would be impossible to generate a response message before the request has been processed.

Creating a broker instance

You can use Apache ActiveMQ as the JMS implementation. A convenient approach to use in this demonstration is to embed the Apache ActiveMQ broker in the bridge bundle. Simply define an **`amq:broker`** element in the Spring XML file, as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      ...
      xmlns:amq="http://activemq.apache.org/schema/core"
      ...>

  <amq:broker brokerName="CxfPayloadDemo" persistent="false">
      <amq:transportConnectors>
```

```
          <amq:transportConnector name="openwire"
uri="tcp://localhost:51616"/>
          <amq:transportConnector name="vm" uri="vm:local"/>
      </amq:transportConnectors>
  </amq:broker>
  ...
</beans>
```



NOTE

This broker instance is created with the **`persistent`** attribute set to **`false`**, so that the messages are stored *only* in memory.

Configuring the JMS component

Because the broker is co-located with the bridge route (in the same JVM), the most efficient way to connect to the broker is to use the VM (Virtual Machine) transport. Configure the Apache ActiveMQ component as follows, to connect to the co-located broker using the VM protocol:

```
<beans ...>
    ...
    <bean id="activemq"
class="org.apache.activemq.camel.component.ActiveMQComponent">
        <property name="brokerURL" value="vm:local"/>
    </bean>
    ...
</beans>
```

NOTE

By defining the bean with an **`id`** value of **`activemq`**, you are implicitly overriding the component associated with the endpoint URI prefix, **`activemq:`**. In other words, your custom **`ActiveMQComponent`** instance is used instead of the default **`ActiveMQComponent`** instance from the **`camel-activemq`** JAR file.

Sample SOAP/HTTP-to-JMS route

For example, you could define a route that implements the SOAP/HTTP-to-JMS bridge specifically for the **`updateCustomer`** operation from the **`CustomerService`** SEI, as follows:

```
<when>
    <simple>${in.header.operationName} == 'updateCustomer'</simple>
    <log message="Placing update customer message onto queue."/>
    <inOnly uri="activemq:queue:CustomerUpdates?jmsMessageType=Text"/>
    <transform>
        <constant>
        <![CDATA[
<ns2:updateCustomerResponse
xmlns:ns2="http://demo.fusesource.com/wsdl/CustomerService/"/>
        ]]>
        </constant>
    </transform>
</when>
```

Sending to the JMS endpoint in inOnly mode

Note how the message payload is sent to the JMS queue using the **`inOnly`** DSL command instead of the **`to`** DSL command. When you send a message using the **`to`** DSL command, the default behavior is to use the same invocation mode as the current exchange. But the current exchange has an *InOut* MEP, which means that the **`to`** DSL command would wait forever for a response message from JMS.

The invocation mode we want to use when sending the payload to the JMS queue is *InOnly* (asynchronous), and we can force this mode by inserting the **`inOnly`** DSL command into the route.

NOTE

By specifying the option, **`jmsMessageType=Text`**, Camel CXF implicitly converts the message payload to an XML string before pushing it onto the JMS queue.

Returning a literal response value

The **`transform`** DSL command uses an expression to set the body of the exchange's *Out* message and this message is then used as the response to the client. Your first impulse when defining a response in XML format might be to use a DOM API, but in this example, the response is specified as a string literal. This approach has the advantage of being both efficient and very easy to program.

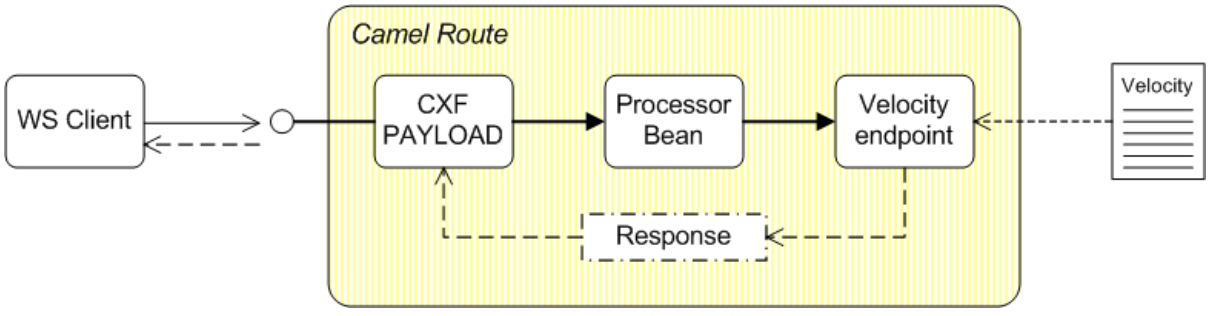
The final step of processing, which consists of converting the XML string to a DOM object, is performed by Apache Camel's implicit type conversion mechanism.

6.5. GENERATING RESPONSES USING TEMPLATES

Overview

One of the simplest and quickest approaches to generating a response message is to use a velocity template. Figure 6.3, “Response Generated by Velocity” shows the outline of a general template-based route. At the start of the route is a Camel CXF endpoint in PAYLOAD mode, which is the appropriate mode to use for processing the message as an XML document. After doing the work required to process the message and stashing some intermediate results in message headers, the route generates the response message using a Velocity template.

Figure 6.3. Response Generated by Velocity



Sample template-based route

For example, you could define a template-based route specifically for the **`getCustoemrStatus`** operation, as follows:

```
. . .
```

```
        <when>
          <simple>${in.header.operationName} ==
'getCustomerStatus'</simple>
          <convertBodyTo type="org.w3c.dom.Node"/>
          <setHeader headerName="customerId">
            <xpath
resultType="java.lang.String">/cus:getCustomerStatus/customerId</xpath>
          </setHeader>
          <to uri="getCustomerStatus"/>
          <to uri="velocity:getCustomerStatusResponse.vm"/>
        </when>
      </choice>
    </route>
  </camelContext
  ...
  <bean id="getCustomerStatus"
    class="com.fusesource.customerwscamelcxfpayload.GetCustomerStatus"/>
```

Route processing steps

Given the preceding route definition, any message whose operation name matches **`getCustomerStatus`** would be processed as follows:

1. To facilitate processing the payload body, the first step uses **`convertBodyTo`** to convert the body type from **`org.apache.camel.component.cxf.CxfPayload`** (the default payload type) to **`org.w3c.dom.Node`**.

2. The route then applies an XPath expression to the message in order to extract the customer ID value and then stashes it in the **`customerId`** header.

3. The next step sends the message to the **`getCustomerStatus`** bean, which does whatever processing is required to get the customer status for the specified customer ID. The results from this step are stashed in message headers.

4. Finally, a response is generated using a velocity template.

NOTE

A common pattern when implementing Apache Camel routes is to use message headers as a temporary stash to hold intermediate results (you could also use exchange properties in the same way).

Converting XPath result to a string

Because the default return type of XPath is a node list, you must explicitly convert the result to a string, if you want to obtain the string contents of an element. There are two alternative ways of obtaining the string value of an element:

- Specify the result type explicitly using the **`resultType`** attribute, as follows:

  ```
  <xpath
  resultType="java.lang.String">/cus:getCustomerStatus/customerId</xpa
  th>
  ```

- Modify the expression so that it returns a **text()** node, which automatically converts to string:

```
<xpath>/cus:getCustomerStatus/customerId/text()</xpath>
```

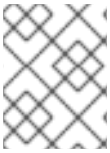
getCustomerStatus processor bean

The **getCustomerStatus** processor bean is an instance of the **GetCustomerStatus** processor class, which is defined as follows:

```
// Java
package com.fusesource.customerwscamelcxfpayload;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;

public class GetCustomerStatus implements Processor
{
    public void process(Exchange exchng) throws Exception {
        String id = exchng.getIn().getHeader("customerId", String.class);

        // Maybe do some kind of lookup here!
        //

        exchng.getIn().setHeader("status", "Away");
        exchng.getIn().setHeader("statusMessage", "Going to sleep.");
    }
}
```

The implementation shown here is just a placeholder. In a realistic application you would perform some sort of checks or database lookup to obtain the customer status. In the demonstration code, however, the **status** and **statusMessage** are simply set to constant values and stashed in message headers.

In the preceding code, we make the modifications directly to the *In* message. When the exchange's *Out* message is **null**, the next processor in the route gets a copy of the current *In* message instead



NOTE

An exceptional case occurs when the message exchange pattern is *inOnly*, in which case the *Out* message value is *always* copied into the In message, even if it is **null**.

getCustomerStatusResponse.vm Velocity template

You can generate a response message very simply using a Velocity template. The Velocity template consists of a message in plain text, where specific pieces of data can be inserted using expressions—for example, the expression **${header.*HeaderName*}** substitutes the value of a named header.

The Velocity template for generating the **getCustomerStatus** reponse is located in the **customer-ws-camel-cxf-payload/src/main/resources** directory and it contains the following template script:

```
<ns2:getCustomerStatusResponse
xmlns:ns2="http://demo.fusesource.com/wsdl/CustomerService/">
   <status>${headers.status}</status>
```

```
    <statusMessage>${headers.statusMessage}</statusMessage>
</ns2:getCustomerStatusResponse>
```

6.6. DEPLOY TO OSGI

Overview

One of the options for deploying the payload-based route is to package it as an OSGi bundle and deploy it into an OSGi container such as Red Hat JBoss Fuse. Some of the advantages of an OSGi deployment include:

- Bundles are a relatively lightweight deployment option (because dependencies can be shared between deployed bundles).
- OSGi provides sophisticated dependency management, ensuring that only version-consistent dependencies are added to the bundle's classpath.

Using the Maven bundle plug-in

The Maven bundle plug-in is used to package your project as an OSGi bundle, in preparation for deployment into the OSGi container. There are two essential modifications to make to your project's **`pom.xml`** file:

1. Change the packaging type to **`bundle`** (by editing the value of the **`project/packaging`** element in the POM).
2. Add the Maven bundle plug-in to your POM file and configure it as appropriate.

Configuring the Maven bundle plug-in is quite a technical task (although the default settings are often adequate). For full details of how to customize the plug-in configuration, consult *Deploying into the OSGi Container* and *Managing OSGi Dependencies*.

Sample bundle plug-in configuration

The following POM fragment shows a sample configuration of the Maven bundle plug-in, which is appropriate for the current example.

```
<?xml version="1.0"?>
<project  ...>
  ...
  <groupId>org.fusesource.sparks.fuse-webinars.cxf-webinars</groupId>
  <artifactId>customer-ws-camel-cxf-payload</artifactId>

  <name>customer-ws-camel-cxf-payload</name>
  <url>http://www.fusesource.com</url>
 <packaging>bundle</packaging>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
```

```
        <configuration>
         <instructions>
            <Import-Package>
              org.apache.camel.component.velocity,
              META-INF.cxf,
              META-INF.cxf.osgi,
              javax.jws,
              javax.wsdl,
              javax.xml.bind,
              javax.xml.bind.annotation,
              javax.xml.namespace,
              javax.xml.ws,
              org.w3c.dom,
              <!-- Workaround to access DOM XPathFactory -->
              org.apache.xpath.jaxp,
              *
            </Import-Package>
            <DynamicImport-Package>
              org.apache.cxf.*,
              org.springframework.beans.*
            </DynamicImport-Package>
          </instructions>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
</project>
```

Dynamic imports

The Java packages from Apache CXF and the Spring API are imported using dynamic imports (specified using the **`DynamicImport-Package`** element). This is a pragmatic way of dealing with the fact that Spring XML files are not terribly well integrated with the Maven bundle plug-in. At build time, the Maven bundle plug-in is *not* able to figure out which Java classes are required by the Spring XML code. By listing wildcarded package names in the **`DynamicImport-Package`** element, however, you allow the OSGi container to figure out which Java classes are needed by the Spring XML code at *run* time.

NOTE

In general, using **`DynamicImport-Package`** headers is not recommended in OSGi, because it short-circuits OSGi version checking. Normally, what should happen is that the Maven bundle plug-in lists the Java packages used at *build* time, along with their versions, in the **`Import-Package`** header. At *deploy* time, the OSGi container then checks that the available Java packages are compatible with the build time versions listed in the **`Import-Package`** header. With dynamic imports, this version checking cannot be performed.

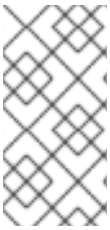
Build and deploy the client bundle

After you have configured the POM file, you can build the Maven project and install it in your local repository by entering the following command:

```
mvn install
```

To deploy the route bundle, enter the following command at the console:

```
karaf@root> install -s mvn:org.fusesource.sparks.fuse-webinars.cxf-
webinars/customer-ws-camel-cxf-payload
```



NOTE

If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **`org.ops4j.pax.url.mvn.localRepository`** property in the ***`EsbInstallDir`*`/etc/org.ops4j.pax.url.mvn.cfg`** file, before you can use the **`mvn:`** scheme to access Maven artifacts.

CHAPTER 7. PROVIDER-BASED ROUTE

7.1. PROVIDER-BASED JAX-WS ENDPOINT

Overview

Use the provider-based approach, if you need to process *very* large Web services messages. The provider-based approach is a variant of the PAYLOAD data format that enables you to encode the message body as an XML streaming type, such as **`SAXSource`**. Since the XMLstreaming types are more efficient than DOM objects, the provider-based approach is ideal for large XML messages.

Demonstration location

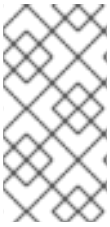
The code presented in this chapter is taken from the following demonstration:

```
DemoDir/src/fuse-webinars/cxf-webinars/customer-ws-camel-cxf-provider
```

For details of how to download and install the demonstration code, see Chapter 1, *Demonstration Code for Camel/CXF*

Camel CXF component

The Camel CXF component is an Apache CXF component that integrates Web services with routes. You can use it either to instantiate *consumer endpoints* (at the start of a route), which behave like Web service instances, or to instantiate *producer endpoints* (at any other points in the route), which behave like WS clients.



NOTE

Came CXF endpoints—which are instantiated using the **`cxf:cxfEndpoint`** XML element and are implemented by the Apache Camel project—are not to be confused with the Apache CXF JAX-WS endpoints—which are instantiated using the **`jaxws:endpoint`** XML element and are implemented by the Apache CXF project.

Provider-based approach and the PAYLOAD data format

The provider-based approach is a variant of the PAYLOAD data format, which is enabled as follows:

- Define a custom **`javax.xml.ws.Provider<StreamType>`** class, where the *StreamType* type is an XML streaming type, such as **`SAXSource`**.
- The PAYLOAD data format is selected by an annotation on the custom **`Provider<?>`** class (see the section called “The SAXSourceService provider class”).
- The custom **`Provider<?>`** class is referenced by setting the **`serviceClass`** attribute of the **`cxf:cxfEndpoint`** element in XML configuration.

The provider-based approach has the following characteristics:

- Enables you to access the message body as a streamed XML type—for example, **`javax.xml.transform.sax.SAXSource`**.

- No JAX-WS or JAX-B stub code required.
- The SOAP body is marshalled into a stream-based **`SAXSource`** type.
- The SOAP headers are converted into headers in the exchange's *In* message, of **`org.apache.cxf.binding.soap.SoapHeader`** type.

Implementing and building a provider-based route

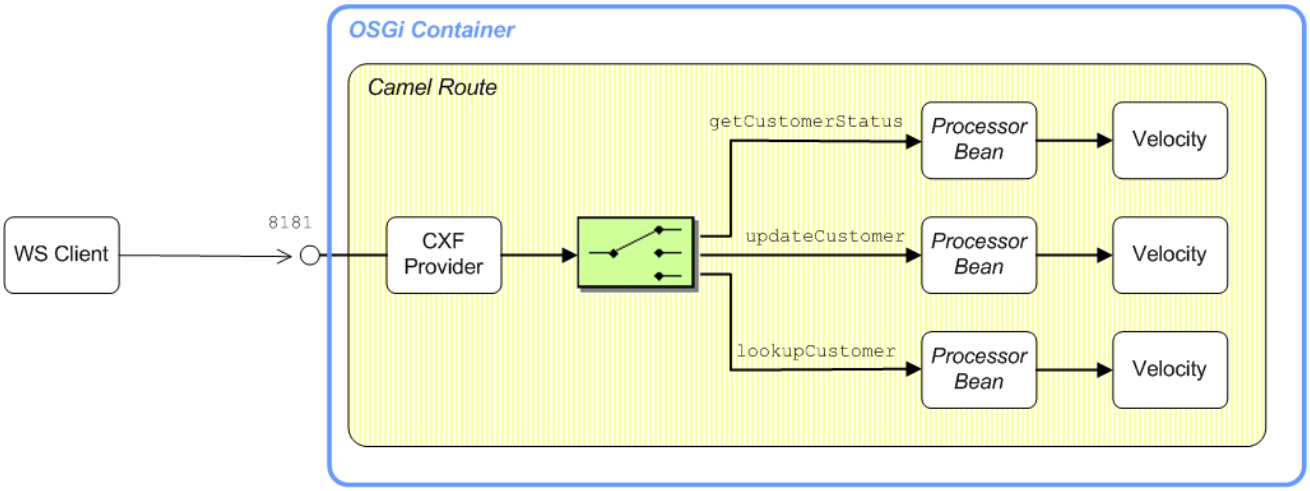
To implement and build the demonstration provider-based route, starting from scratch, you would perform the following steps:

1. Define a custom **`javax.xml.ws.Provider<StreamType>`** class (the current demonstration uses **`SAXSource`** as the *StreamType* type).
2. Instantiate the Camel CXF endpoint in Spring, using the **`cxf:cxfEndpoint`** element and reference the custom provider class (using the **`serviceClass`** attribute).
3. Implement the route in XML, where you can use the content-based router to sort requests by operation name.
4. For each operation, define a processor bean to process the request.
5. Define velocity templates for generating the reponse messages.
6. Define a custom type converter, to support converting a **`String`** message body to a **`SAXSource`** message body.

Sample provider-based route

Figure 7.1, “Sample Provider-Based Route” shows an outline of the route that is used to process the operations of the **`CustomerService`** Web service using the provider-based approach. After sorting the request messages by operation name, an operation-specific processor bean reads the incoming request parameters. Finally, the response messages are generated using Velocity templates.

Figure 7.1. Sample Provider-Based Route



7.2. CREATE A PROVIDER<?> IMPLEMENTATION CLASS

Overview

The fundamental prerequisite for using provider mode is to define a custom **`Provider<>`** class that implements the **`invoke()`** method. In fact, the sole purpose of this class is to provide runtime type information for Apache CXF: *the **`invoke()`** method never gets called!*

By implementing the provider class in the way shown here, you are merely indicating to the Apache CXF runtime that the WS endpoint should operate in in PAYLOAD mode and the type of the message PAYLOAD should be **`SAXSource`**.

The SAXSourceService provider class

The definition of the provider class is relatively short and the complete definition of the customer provider class, **`SAXSourceService`**, is as follows:

```
// Java
package com.fusesource.customerwscamelcxfprovider;

import javax.xml.transform.sax.SAXSource;
import javax.xml.ws.Provider;
import javax.xml.ws.Service.Mode;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider()
@ServiceMode(Mode.PAYLOAD)
public class SAXSourceService implements Provider<SAXSource>
{
  public SAXSource invoke(SAXSource t) {
    throw new UnsupportedOperationException("Not supported yet.");
  }
}
```

The customer provider class, **`SAXSourceService`**, must be annotated by the **`@WebServiceProvider`** annotation to mark it as a provider class and can be optionally annotated by the **`@ServiceMode`** annotation to select PAYLOAD mode.

7.3. INSTANTIATE THE WS ENDPOINT

Overview

In Apache Camel, the CXF component is the key to integrating routes with Web services. You can use the CXF component to create two different kinds of endpoint:

- *Consumer endpoint*—(at the start of a route) represents a Web service instance, which integrates with the route. The type of payload injected into the route depends on the value of the endpoint's **`dataFormat`** option.
- *Producer endpoint*—represents a special kind of WS client proxy, which converts the current exchange object into an operation invocation on a remote Web service. The format of the current exchange must match the endpoint's **`dataFormat`** setting.

The cxf:bean: URI syntax

The **cxf:bean:** URI is used to bind an Apache CXF endpoint to a route and has the following general syntax:

```
cxf:bean:CxfEndpointID[?Options]
```

Where ***CxfEndpointID*** is the ID of a bean created using the **cxf:cxfEndpoint** element, which configures the details of the WS endpoint. You can append options to this URI (where the options are described in detail in chapter "CXF" in "EIP Component Reference"). Provider mode is essentially a variant of PAYLOAD mode: you could specify this mode on the URI (by setting **dataFormat=PAYLOAD**), but this is not necessary, because PAYLOAD mode is already selected by the **@ServiceMode** annotation on the custom **Provider** class.

For example, to start a route with an endpoint in provider mode, where the endpoint is configured by the **customer-ws** bean, define the route as follows:

```
<route>
    <from uri="cxf:bean:customer-ws"/>
    ...
</route>
```

The cxf:cxfEndpoint element

The **cxf:cxfEndpoint** element is used to define a WS endpoint that binds either to the start (consumer endpoint) or the end (producer endpoint) of a route. For example, to define the **customer-ws** WS endpoint in provider mode, you define a **cxf:cxfEndpoint** element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    ...
    <cxf:cxfEndpoint id="customer-ws"
        address="/Customer"
        endpointName="c:SOAPOverHTTP"
        serviceName="c:CustomerService"
        wsdlURL="wsdl/CustomerService.wsdl"

serviceClass="com.fusesource.customerwscamelcxfprovider.SAXSourceService"
        xmlns:c="http://demo.fusesource.com/wsdl/CustomerService/"/>
    ...
</beans>
```

Specifying the WSDL location

The **wsdlURL** attribute of the **cxf:cxfEndpoint** element is used to specify the location of the WSDL contract for this endpoint. The WSDL contract is used as the source of metadata for this endpoint.

Specifying the service class

A key difference between provider mode and ordinary PAYLOAD mode is that the **serviceClass** attribute must be set to the provider class, **SAXSourceService**.

7.4. SORT MESSAGES BY OPERATION NAME

The operationName header

When the WS endpoint parses an incoming operation invocation in PROVIDER mode, it automatically sets the **operationName** header to the name of the invoked operation. You can then use this header to sort messages by operation name.

Sorting by operation name

For example, the **customer-ws-camel-cxf-provider** demonstration defines the following route, which uses the content-based router pattern to sort incoming messages, based on the operation name. The **when** predicates check the value of the **operationName** header using simple language expressions, sorting messages into invocations on the **updateCustomer** operation, the **lookupCustomer** operation, or the **getCustomerStatus** operation.

```
<beans ...>
    ...
    <camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="cxf:bean:customer-ws"/>
            <choice>
                <when>
                    <simple>${in.header.operationName} ==
'updateCustomer'</simple>
                    ...
                </when>
                <when>
                    <simple>${in.header.operationName} ==
'lookupCustomer'</simple>
                    ...
                </when>
                <when>
                    <simple>${in.header.operationName} ==
'getCustomerStatus'</simple>
                    ...
                </when>
            </choice>
        </route>
    </camelContext>
    ...
</beans>
```

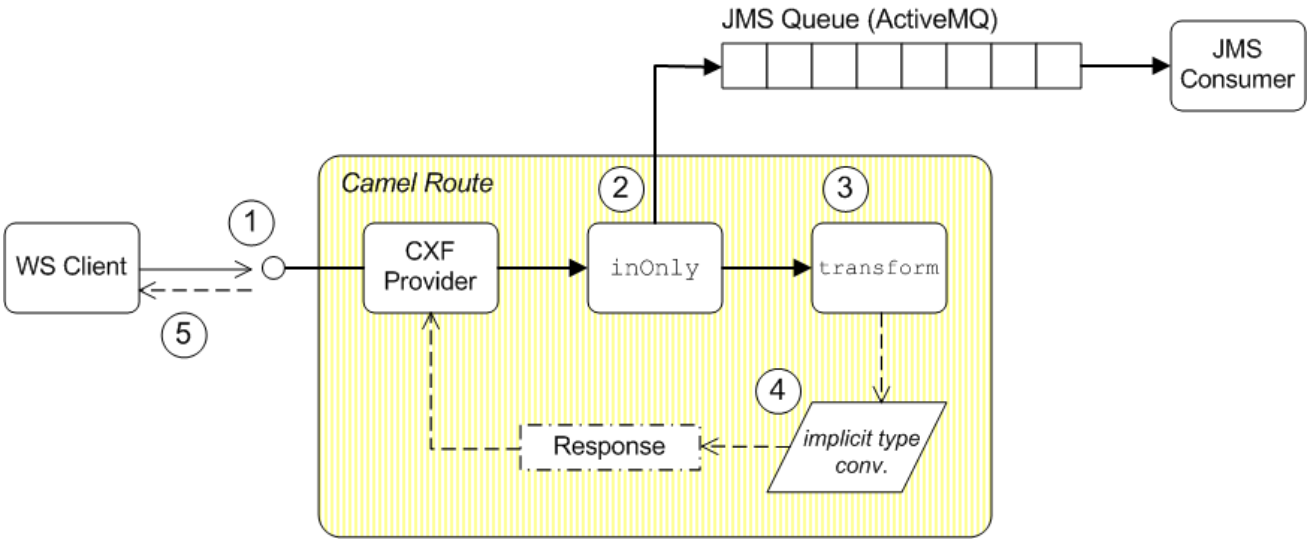
7.5. SOAP/HTTP-TO-JMS BRIDGE USE CASE

Overview

In this section, we consider a SOAP/HTTP-to-JMS bridge use case: that is, you want to create a route that transforms a synchronous operation invocation (over SOAP/HTTP) into an asynchronous message delivery (by pushing the message onto a JMS queue). In this way, it becomes possible to process the incoming operation invocations at a later time, by pulling messages off the JMS queue.

Figure 7.2, “SOAP/HTTP-to-JMS Bridge” shows the general outline of a bridge that can transform synchronous SOAP/HTTP invocations into asynchronous JMS message deliveries.

Figure 7.2. SOAP/HTTP-to-JMS Bridge



Transforming RPC operations to One Way

As shown in Figure 7.2, “SOAP/HTTP-to-JMS Bridge”, the route for transforming synchronous SOAP/HTTP to asynchronous JMS works as follows:

1. The WS client invokes a synchronous operation on the Camel CXF endpoint at the start of the route. The Camel CXF endpoint then creates an initial *InOut* exchange at the start of the route, where the body of the exchange message contains a payload in XML format.
2. The **`inOnly`** DSL command pushes a copy of the XML payload onto a JMS queue, so that it can be processed offline at some later time.
3. The **`transform`** DSL command constructs an immediate response to send back to the client, where the response has the form of an XML string.
4. The route explicitly converts the XML string to the **`javax.xml.transform.sax.SAXSource`** type.
5. The response is sent back to the WS client, thus completing the synchronous operation invocation.

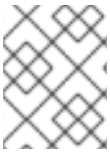
Evidently, this transformation can only work, if the original operation invocation has no return value. Otherwise, it would be impossible to generate a response message before the request has been processed.

Creating a broker instance

You can use Apache ActiveMQ as the JMS implementation. A convenient approach to use in this demonstration is to embed the Apache ActiveMQ broker in the bridge bundle. Simply define an **`amq:broker`** element in the Spring XML file, as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      ...
      xmlns:amq="http://activemq.apache.org/schema/core"
      ...>

  <amq:broker brokerName="CxfPayloadDemo" persistent="false">
      <amq:transportConnectors>
```

```
          <amq:transportConnector name="openwire"
uri="tcp://localhost:51616"/>
          <amq:transportConnector name="vm" uri="vm:local"/>
      </amq:transportConnectors>
  </amq:broker>
  ...
</beans>
```



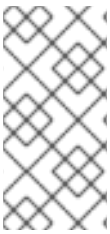
NOTE

This broker instance is created with the **`persistent`** attribute set to **`false`**, so that the messages are stored *only* in memory.

Configuring the JMS component

Because the broker is co-located with the bridge route (in the same JVM), the most efficient way to connect to the broker is to use the VM (Virtual Machine) transport. Configure the Apache ActiveMQ component as follows, to connect to the co-located broker using the VM protocol:

```
<beans ...>
    ...
    <bean id="activemq"
class="org.apache.activemq.camel.component.ActiveMQComponent">
        <property name="brokerURL" value="vm:local"/>
    </bean>
    ...
</beans>
```



NOTE

By defining the bean with an **`id`** value of **`activemq`**, you are implicitly overriding the component associated with the endpoint URI prefix, **`activemq:`**. In other words, your custom **`ActiveMQComponent`** instance is used instead of the default **`ActiveMQComponent`** instance from the **`camel-activemq`** JAR file.

Sample SOAP/HTTP-to-JMS route

For example, you could define a route that implements the SOAP/HTTP-to-JMS bridge specifically for the **`updateCustomer`** operation from the **`CustomerService`** SEI, as follows:

```
<when>
    <simple>${in.header.operationName} == 'updateCustomer'</simple>
    <log message="Placing update customer message onto queue."/>
    <inOnly uri="activemq:queue:CustomerUpdates?jmsMessageType=Text"/>
    <transform>
        <constant>
        <![CDATA[
<ns2:updateCustomerResponse
xmlns:ns2="http://demo.fusesource.com/wsdl/CustomerService/"/>
        ]]>
        </constant>
```

```
    </transform>
    <convertBodyTo type="javax.xml.transform.sax.SAXSource"/>
</when>
```

Sending to the JMS endpoint in inOnly mode

Note how the message payload is sent to the JMS queue using the **inOnly** DSL command instead of the **to** DSL command. When you send a message using the **to** DSL command, the default behavior is to use the same invocation mode as the current exchange. But the current exchange has an *InOut* MEP, which means that the **to** DSL command would wait forever for a response message from JMS.

The invocation mode we want to use when sending the payload to the JMS queue is *InOnly* (asynchronous), and we can force this mode by inserting the **inOnly** DSL command into the route.

NOTE

By specifying the option, **jmsMessageType=Text**, Camel CXF implicitly converts the message payload to an XML string before pushing it onto the JMS queue.

Returning a literal response value

The **transform** DSL command uses an expression to set the body of the exchange's *Out* message and this message is then used as the response to the client. Your first impulse when defining a response in XML format might be to use a DOM API, but in this example, the response is specified as a string literal. This approach has the advantage of being both efficient and very easy to program.

The final step of processing, which consists of converting the XML string to a DOM object, is performed by Apache Camel's implicit type conversion mechanism.

Type conversion of the response message

In this example, the reply message (like the request message) is required to be of type, **javax.xml.transform.sax.SAXSource**. In the last step of the route, therefore, you must convert the message body from **String** type to **javax.xml.transform.sax.SAXSource** type, by invoking the **convertBodyTo** DSL command.

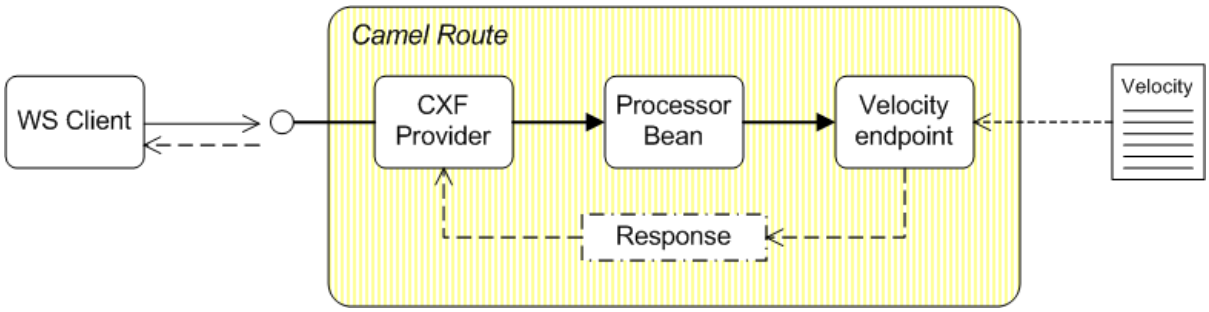
The implementation of the String to **SAXSource** conversion is provided by a custom type converter, as described in Section 7.7, “TypeConverter for SAXSource”.

7.6. GENERATING RESPONSES USING TEMPLATES

Overview

One of the simples and quickest approaches to generating a response message is to use a velocity template. Figure 7.3, “Response Generated by Velocity” shows the outline of a general template-based route. At the start of the route is a Camel CXF endpoint in provider mode, which is the appropriate mode to use for processing the message as an XML document. After doing the work required to process the message and stashing some intermediate results in message headers, the route generates the response message using a Velocity template.

Figure 7.3. Response Generated by Velocity



Sample template-based route

For example, you could define a template-based route specifically for the **`getCustoemrStatus`** operation, as follows:

```
        ...
        <when>
          <simple>${in.header.operationName} ==
'getCustomerStatus'</simple>
          <setHeader headerName="customerId">
            <xpath
resultType="java.lang.String">/cus:getCustomerStatus/customerId</xpath>
          </setHeader>
          <to uri="getCustomerStatus"/>
          <to uri="velocity:getCustomerStatusResponse.vm"/>
          <convertBodyTo type="javax.xml.transform.sax.SAXSource"/>
        </when>
      </choice>
    </route>
  </camelContext
  ...
  <bean id="getCustomerStatus"
    class="com.fusesource.customerwscamelcxfpayload.GetCustomerStatus"/>
```

Route processing steps

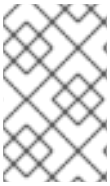
Given the preceding route definition, any message whose operation name matches **`getCustomerStatus`** would be processed as follows:

1. The route applies an XPath expression to the message in order to extract the customer ID value and then stashes it in the **`customerId`** header.

2. The next step sends the message to the **`getCustomerStatus`** bean, which does whatever processing is required to get the customer status for the specified customer ID. The results from this step are stashed in message headers.

3. A response is generated using a Velocity template.

4. Finally, the XML string generated by the Velocity template must be explicitly converted to the **`javax.xml.transform.sax.SAXSource`** type using **`convertBodyTo`** (which implicitly relies on a type converter).



NOTE

A common pattern when implementing Apache Camel routes is to use message headers as a temporary stash to hold intermediate results (you could also use exchange properties in the same way).

XPath expressions and SAXSource

XPath expressions can be applied directly to SAXSource objects. The XPath implementation has a pluggable architecture that supports a variety of XML parsers and when XPath encounters a SAXSource object, it automatically loads the plug-in required to support SAXSource parsing.

getCustomerStatus processor bean

The **`getCustomerStatus`** processor bean is an instance of the **`GetCustomerStatus`** processor class, which is defined as follows:

```
// Java
package com.fusesource.customerwscamelcxfpayload;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;

public class GetCustomerStatus implements Processor
{
    public void process(Exchange exchng) throws Exception {
        String id = exchng.getIn().getHeader("customerId", String.class);

        // Maybe do some kind of lookup here!
        //

        exchng.getIn().setHeader("status", "Away");
        exchng.getIn().setHeader("statusMessage", "Going to sleep.");
    }
}
```

The implementation shown here is just a placeholder. In a realistic application you would perform some sort of checks or database lookup to obtain the customer status. In the demonstration code, however, the **`status`** and **`statusMessage`** are simply set to constant values and stashed in message headers.

getCustomerStatusResponse.vm Velocity template

You can generate a response message very simply using a Velocity template. The Velocity template consists of a message in plain text, where specific pieces of data can be inserted using expressions—for example, the expression **`${header.`*`HeaderName`*`}`** substitutes the value of a named header.

The Velocity template for generating the **`getCustomerStatus`** reponse is located in the **`customer-ws-camel-cxf-provider/src/main/resources`** directory and it contains the following template script:

```
<ns2:getCustomerStatusResponse
xmlns:ns2="http://demo.fusesource.com/wsdl/CustomerService/">
   <status>${headers.status}</status>
```

```
    <statusMessage>${headers.statusMessage}</statusMessage>
</ns2:getCustomerStatusResponse>
```

7.7. TYPECONVERTER FOR SAXSOURCE

Overview

Apache Camel supports a type converter mechanism, which is used to perform implicit and explicit type conversions of message bodies and message headers. The type converter mechanism is extensible and it so happens that the provider demonstration requires a custom type converter that can convert **`String`** objects to **`SAXSource`** objects.

String to SAXSource type converter

The String to SAXSource type converter is implemented in the **`AdditionalConverters`** class, as follows:

```
// Java
package com.fusesource.customerwscamelcxfprovider;

import java.io.ByteArrayInputStream;
import javax.xml.transform.sax.SAXSource;
import org.apache.camel.Converter;
import org.xml.sax.InputSource;

@Converter
public class AdditionalConverters {

  @Converter
  public static SAXSource toSAXSource(String xml) {
      return new SAXSource(new InputSource(new
ByteArrayInputStream(xml.getBytes())));
  }
}
```

Reference

For full details of the type converter mechanism in Apache Camel, see section "Built-In Type Converters" in "Programming EIP Components" and chapter "Type Converters" in "Programming EIP Components".

7.8. DEPLOY TO OSGI

Overview

One of the options for deploying the provider-based route is to package it as an OSGi bundle and deploy it into an OSGi container such as Red Hat JBoss Fuse. Some of the advantages of an OSGi deployment include:

- Bundles are a relatively lightweight deployment option (because dependencies can be shared between deployed bundles).

- OSGi provides sophisticated dependency management, ensuring that only version-consistent dependencies are added to the bundle's classpath.

Using the Maven bundle plug-in

The Maven bundle plug-in is used to package your project as an OSGi bundle, in preparation for deployment into the OSGi container. There are two essential modifications to make to your project's **`pom.xml`** file:

1. Change the packaging type to **`bundle`** (by editing the value of the **`project/packaging`** element in the POM).
2. Add the Maven bundle plug-in to your POM file and configure it as appropriate.

Configuring the Maven bundle plug-in is quite a technical task (although the default settings are often adequate). For full details of how to customize the plug-in configuration, consult *Deploying into the OSGi Container* and *Managing OSGi Dependencies*.

Sample bundle plug-in configuration

The following POM fragment shows a sample configuration of the Maven bundle plug-in, which is appropriate for the current example.

```
<?xml version="1.0"?>
<project  ...>
  ...
  <groupId>org.fusesource.sparks.fuse-webinars.cxf-webinars</groupId>
  <artifactId>customer-ws-camel-cxf-provider</artifactId>

  <name>customer-ws-camel-cxf-provider</name>
  <url>http://www.fusesource.com</url>
 <packaging>bundle</packaging>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
         <instructions>
            <Import-Package>
              org.apache.camel.component.velocity,
              META-INF.cxf,
              META-INF.cxf.osgi,
              javax.jws,
              javax.wsdl,
              javax.xml.bind,
              javax.xml.bind.annotation,
              javax.xml.namespace,
              javax.xml.ws,
              org.w3c.dom,
              <!-- Workaround to access DOM XPathFactory -->
              org.apache.xpath.jaxp,
```

```
            *
          </Import-Package>
          <DynamicImport-Package>
            org.apache.cxf.*,
            org.springframework.beans.*
          </DynamicImport-Package>
        </instructions>
      </configuration>
    </plugin>
    ...
  </plugins>
 </build>
</project>
```

Dynamic imports

The Java packages from Apache CXF and the Spring API are imported using dynamic imports (specified using the **`DynamicImport-Package`** element). This is a pragmatic way of dealing with the fact that Spring XML files are not terribly well integrated with the Maven bundle plug-in. At build time, the Maven bundle plug-in is *not* able to figure out which Java classes are required by the Spring XML code. By listing wildcarded package names in the **`DynamicImport-Package`** element, however, you allow the OSGi container to figure out which Java classes are needed by the Spring XML code at *run* time.

NOTE

In general, using **`DynamicImport-Package`** headers is not recommended in OSGi, because it short-circuits OSGi version checking. Normally, what should happen is that the Maven bundle plug-in lists the Java packages used at *build* time, along with their versions, in the **`Import-Package`** header. At *deploy* time, the OSGi container then checks that the available Java packages are compatible with the build time versions listed in the **`Import-Package`** header. With dynamic imports, this version checking cannot be performed.

Build and deploy the client bundle

After you have configured the POM file, you can build the Maven project and install it in your local repository by entering the following command:

```
mvn install
```

To deploy the route bundle, enter the following command at the container console:

```
karaf@root> install -s mvn:org.fusesource.sparks.fuse-webinars.cxf-
webinars/customer-ws-camel-cxf-provider
```

NOTE

If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **`org.ops4j.pax.url.mvn.localRepository`** property in the ***`EsbInstallDir`*`/etc/org.ops4j.pax.url.mvn.cfg`** file, before you can use the **`mvn:`** scheme to access Maven artifacts.

CHAPTER 8. PROXYING A WEB SERVICE

Abstract

A common use case for the Camel CXF component is to use a route as a *proxy* for a Web service. That is, in order to perform additional processing of WS request and response messages, you interpose a route between the WS client and the original Web service.

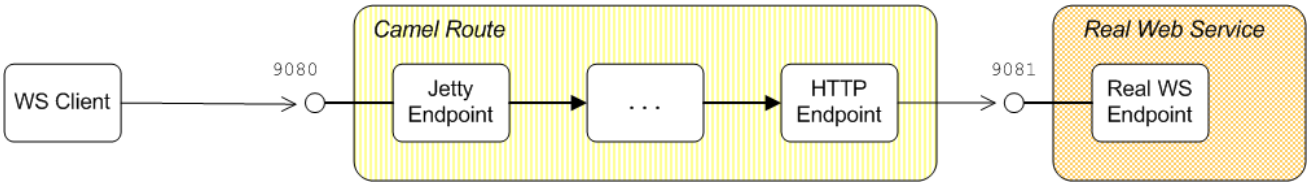
8.1. PROXYING WITH HTTP

Overview

The simplest way to proxy a SOAP/HTTP Web service is to treat the request and reply messages as HTTP packets. This type of proxying can be used where there is no requirement to read or modify the messages passing through the route. For example, you could use this kind of proxying to apply various patterns of flow control on the WS messges.

Figure 8.1, “Proxy Route with Message in HTTP Format” shows an overview of how to proxy a Web service using an Apache Camel route, where the route treats the messages as HTTP packets. The key feature of this route is that both the consumer endpoint (at the start of the route) and the producer endpoint (at the end of the route) must be compatible with the HTTP packet format.

Figure 8.1. Proxy Route with Message in HTTP Format



Alternatives for the consumer endpoint

The following Apache Camel endpoints can be used as consumer endpoints for HTTP format messages:

- *Jetty endpoint*—is a lightweight Web server. You can use Jetty to handle messages for *any* HTTP-based protocol, including the commonly-used Web service SOAP/HTTP protocol.
- *Camel CXF endpoint in MESSAGE mode*—when a Camel CXF endpoint is used in MESSAGE mode, the body of the exchange message is the raw message received from the transport layer (which is HTTP). In other words, the Camel CXF endpoint in MESSAGE mode is equivalent to a Jetty endpoint in the case of HTTP-based protocols.

Consumer endpoint for HTTP

A Jetty endpoint has the general form, **`jetty:HttpAddress`**. To configure the Jetty endpoint to be a proxy for a Web service, use a **`HttpAddress`** value that is almost identical to the HTTP address the client connects to, except that Jetty's version of **`HttpAddress`** uses the special hostname, **`0.0.0.0`** (which matches *all* of the network interfaces on the current machine).

```
<route>
    <from uri="jetty:http://0.0.0.0:9093/Customers?
matchOnUriPrefix=true"/>
```

```
    ...
</route>
```

matchOnUriPrefix option

Normally, a Jetty consumer endpoint accepts only an *exact* match on the context path. For example, a request that is sent to the address **`http://localhost:9093/Customers`** would be accepted, but a request sent to **`http://localhost:9093/Customers/Foo`** would be rejected. By setting **`matchOnUriPrefix`** to **`true`**, however, you enable a kind of wildcarding on the context path, so that any context path prefixed by **`/Customers`** is accepted.

Alternatives for the producer endpoint

The following Apache Camel endpoints can be used as producer endpoints for HTTP format messages:

- *Jetty HTTP client endpoint—(recommended)* the Jetty library implements a HTTP client. In particular, the Jetty HTTP client features support for **`HttpClient`** thread pools, which means that the Jetty implementation scales particularly well.
- *HTTP endpoint*—the HTTP endpoint implements a HTTP client based on the **`HttpClient`** 3.x API.
- *HTTP4 endpoint*—the HTTP endpoint implements a HTTP client based on the **`HttpClient`** 4.x API.

Producer endpoint for HTTP

To configure a Jetty HTTP endpoint to send HTTP requests to a remote SOAP/HTTP Web service, set the **`uri`** attribute of the **`to`** element at the end of the route to be the address of the remote Web service, as follows:

```
<route>
    ...
    <to uri="jetty:http://localhost:8083/Customers?
bridgeEndpoint=true&amp;throwExceptionOnFailure=false"/>
</route>
```

bridgeEndpoint option

The HTTP component supports a **`bridgeEndpoint`** option, which you can enable on a HTTP producer endpoint to configure the endpoint appropriately for operating in a HTTP-to-HTTP bridge (as is the case in this demonstration). In particular, when **`bridgeEndpoint=true`**, the HTTP endpoint ignores the value of the **`Exchange.HTTP_URI`** header, using the HTTP address from the endpoint URI instead.

throwExceptionOnFailure option

Setting **`throwExceptionOnFailure`** to **`false`** ensures that any HTTP exceptions are relayed back to the original WS client, instead of being thrown within the route.

Handling message headers

When defining a HTTP bridge application, the **`CamelHttp*`** headers set by the consumer endpoint at

the start of the route can affect the behavior of the producer endpoint. For this reason, in a bridge application it is advisable to remove the **`CamelHttp*`** headers before the message reaches the producer endpoint, as follows:

```
<route>
    <from uri="jetty:http:..."/>
    ...
 <removeHeaders pattern="CamelHttp*"/>
    <to uri="jetty:http:..."/>
</route>
```

Outgoing HTTP headers

By default, any headers in the exchange that are *not* prefixed by **`Camel`** will be converted into HTTP headers and sent out over the wire by the HTTP producer endpoint. This could have adverse consequences on the behavior of your application, *so it is important to be aware of any headers that are set in the exchange object and to remove them, if necessary*.

For more details about dealing with headers, see Section 8.4, “Handling HTTP Headers”.

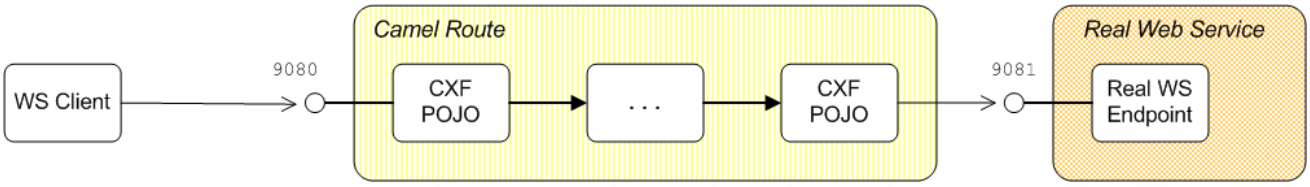
8.2. PROXYING WITH POJO FORMAT

Overview

If you want to access the content of the Web services messages that pass throught the route, you might prefer to process the messages in POJO format: that is, where the body of the exchange consists of a list of Java objects representing the WS operation parameters. The key advantate of using POJO format is that you can easily *process the contents of a message*, by accessing the operation parameters as Java objects.

Figure 8.2, “Proxy Route with Message in POJO Format” shows an overview of how to proxy a Web service using an Apache Camel route, where the route processes the messages in POJO format. The key feature of this route is that both the consumer endpoint (at the start of the route) and the producer endpoint (at the end of the route) must be compatible with the POJO data format.

Figure 8.2. Proxy Route with Message in POJO Format



Consumer endpoint for CXF/POJO

To parse incoming messages into POJO data format, the consumer endpoint at the start of the route must be a Camel CXF endpoint that is configured to use POJO mode. Use the **`cxf:bean:`*`BeanID`*** URI format to reference the Camel CXF endpoint as follows (where the **`dataFormat`** option defaults to POJO):

```
<route>
    <from uri="cxf:bean:customerServiceProxy"/>
    ...
</route>
```

The bean with the ID, **customerServiceProxy**, is a Camel CXF/POJO endpoint, which is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ...
  <cxf:cxfEndpoint
      id="customerServiceProxy"
      xmlns:c="http://demo.fusesource.org/wsdl/camelcxf"
      address="/Customers"
      endpointName="c:SOAPOverHTTP"
      serviceName="c:CustomerService"
      wsdlURL="wsdl/CustomerService.wsdl"
      serviceClass="org.fusesource.demo.wsdl.camelcxf.CustomerService"
  />
  ...
</beans>
```

Producer endpoint for CXF/POJO

To convert the exchange body from POJO data format to a SOAP/HTTP message, the producer endpoint at the end of the route must be a Camel CXF endpoint configured to use POJO mode. Use the **cxf:bean:*BeanID*** URI format to reference the Camel CXF endpoint as follows (where the **dataFormat** option defaults to POJO):

```
<route>
    ...
    <to uri="cxf:bean:customerServiceReal"/>
</route>
```

The bean with the ID, **customerServiceReal**, is a Camel CXF/POJO endpoint, which is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ...
  <cxf:cxfEndpoint
      id="customerServiceReal"
      xmlns:c="http://demo.fusesource.org/wsdl/camelcxf"
      address="http://localhost:8083/Customers"
      endpointName="c:SOAPOverHTTP"
      serviceName="c:CustomerService"
      wsdlURL="wsdl/CustomerService.wsdl"
      serviceClass="org.fusesource.demo.wsdl.camelcxf.CustomerService"
  />
  ...
</beans>
```

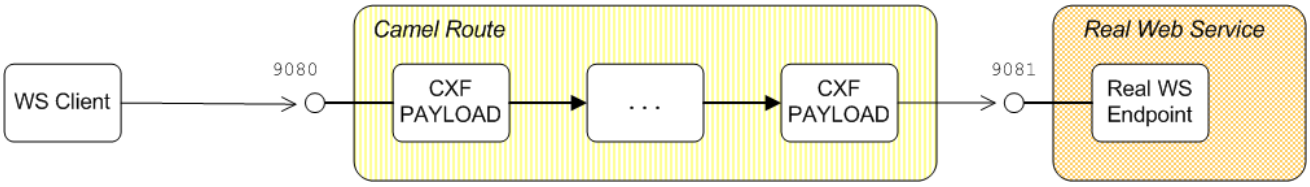
8.3. PROXYING WITH PAYLOAD FORMAT

Overview

If you want to access the content of the Web services messages that pass throught the route, you might prefer to process the messages in the normal PAYLOAD format: that is, where the body of the exchange is accessible as an XML document (essentially, an **`org.w3c.dom.Node`** object). The key advantate of using PAYLOAD format is that you can easily *process the contents of a message*, by accessing the message body as an XML document.

Figure 8.3, "Proxy Route with Message in PAYLOAD Format" shows an overview of how to proxy a Web service using an Apache Camel route, where the route processes the messages in PAYLOAD format. The key feature of this route is that both the consumer endpoint (at the start of the route) and the producer endpoint (at the end of the route) must be compatible with the PAYLOAD data format.

Figure 8.3. Proxy Route with Message in PAYLOAD Format



Consumer endpoint for CXF/PAYLOAD

To parse incoming messages into PAYLOAD data format, the consumer endpoint at the start of the route must be a Camel CXF endpoint that is configured to use PAYLOAD mode. Use the **`cxf:bean:BeanID`** URI format to reference the Camel CXF endpoint as follows, where you *must* set the **`dataFormat`** option to PAYLOAD:

```
<route>
    <from uri="cxf:bean:customerServiceProxy?dataFormat=PAYLOAD"/>
    ...
</route>
```

The bean with the ID, **`customerServiceProxy`**, is a Camel CXF/PAYLOAD endpoint, which is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ...
  <cxf:cxfEndpoint
      id="customerServiceProxy"
      xmlns:c="http://demo.fusesource.org/wsdl/camelcxf"
      address="/Customers"
      endpointName="c:SOAPOverHTTP"
      serviceName="c:CustomerService"
      wsdlURL="wsdl/CustomerService.wsdl"
  />
  ...
</beans>
```

Producer endpoint for CXF/PAYLOAD

To convert the exchange body from PAYLOAD data format to a SOAP/HTTP message, the producer endpoint at the end of the route must be a Camel CXF endpoint configured to use PAYLOAD mode. Use the **`cxf:bean:BeanID`** URI format to reference the Camel CXF endpoint as follows, where you *must* set the **`dataFormat`** option to PAYLOAD:

```
<route>
    ...
    <to uri="cxf:bean:customerServiceReal?dataFormat=PAYLOAD"/>
</route>
```

The bean with the ID, **customerServiceReal**, is a Camel CXF/PAYLOAD endpoint, which is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ...
  <cxf:cxfEndpoint
      id="customerServiceReal"
      xmlns:c="http://demo.fusesource.org/wsdl/camelcxf"
      address="http://localhost:8083/Customers"
      endpointName="c:SOAPOverHTTP"
      serviceName="c:CustomerService"
      wsdlURL="wsdl/CustomerService.wsdl"
  />
  ...
</beans>
```

Outgoing HTTP headers

By default, any headers in the exchange that are *not* prefixed by **Camel** will be converted into HTTP headers and sent out over the wire by the Camel CXF producer endpoint. This could have adverse consequences on the behavior of your application, *so it is important to be aware of any headers that are set in the exchange object and to remove them, if necessary*.

For more details about dealing with headers, see Section 8.4, “Handling HTTP Headers”.

8.4. HANDLING HTTP HEADERS

Overview

When building bridge applications using HTTP or HTTP-based components, it is important to be aware of how the HTTP-based endpoints process headers. In many cases, internal headers (prefixed by Camel) or other headers can cause unwanted side-effects on your application. It is often necessary to remove or filter out certain headings or classes of headings in your route, in order to ensure that your application behaves as expected.

HTTP-based components

The behavior described in this section affects not just the Camel HTTP component (**camel-http**), but also a number of other HTTP-based components, including:

```
camel-http
camel-http4
camel-jetty
camel-restlet
camel-cxf
```

HTTP consumer endpoint

When a HTTP consumer endpoint receives an incoming message, it creates an *In* message with the following headers:

`CamelHttp*` headers

Several headers with the **`CamelHttp`** prefix are created, which record the status of the incoming message. For details of these internal headers, see HTTP.

HTTP headers

All of the HTTP headers from the original incoming message are mapped to headers on the exchange's *In* message.

URL options *(Jetty only)*

The URL options from the original HTTP request URL are mapped to headers on the exchange's *In* message. For example, given the client request with the URL, **`http://myserver/myserver?orderid=123`**, a Jetty consumer endpoint creates the **`orderid`** header with value **`123`**.

HTTP producer endpoint

When a HTTP producer endoint receives an exchange and converts it to the target message format, it handles the *In* message headers as follows:

`CamelHttp*`

Headers prefixed by **`CamelHttp`** are used to control the behavior of the HTTP producer entpoint. Any headers of this kind are consumed by the HTTP producer endpoint and the endpoint behaves as directed.

`Camel*`

All other headers prefixed by **`Camel`** are presumed to be meant for internal use and are *not* mapped to HTTP headers in the target message (in other words, these headers are ignored).

`*`

All other headers are converted to HTTP headers in the target message, with the exception of the following headers, which are blocked (based on a case-insensitive match):

```
content-length
content-type
cache-control
connection
date
pragma
trailer
transfer-encoding
upgrade
via
warning
```

Implications for HTTP bridge applications

When defining a HTTP bridge application (that is, a route starting with a HTTP consumer endpoint and ending with a HTTP producer endpoint), the **CamelHttp*** headers set by the consumer endpoint at the start of the route can affect the behavior of the producer endpoint. For this reason, in a bridge application it is advisable to remove the **CamelHttp*** headers, as follows:

```
from("http://0.0.0.0/context/path")
  .removeHeaders("CamelHttp*")
   ...
   .to("http://remoteHost/context/path");
```

Setting a custom header filter

If you want to customize the way that a HTTP producer endpoint processes headers, you can define your own customer header filter by defining the **headerFilterStrategy** option on the endpoint URI. For example, to configure a producer endpoint with the **myHeaderFilterStrategy** filter, you could use a URI like the following:

```
http://remoteHost/context/path?
headerFilterStrategy=#myHeaderFilterStrategy
```

Where **myHeaderFilterStrategy** is the bean ID of your custom filter instance.

CHAPTER 9. FILTERING SOAP MESSAGE HEADERS

Abstract

The Camel CXF component supports a flexible header filtering mechanism, which enables you to process SOAP headers, applying different filters according to the header's XML namespace.

9.1. BASIC CONFIGURATION

Overview

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

CxfHeaderFilterStrategy

Header filtering is controlled by the **`CxfHeaderFilterStrategy`** class. Basic configuration of the **`CxfHeaderFilterStrategy`** class involves setting one or more of the following options:

- the section called “relayHeaders option”.
- the section called “relayAllMessageHeaders option”.

relayHeaders option

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

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

In-band headers

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

Out-of-band headers

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

Payload format

The CXF endpoint's payload format affects the filter behavior as follows:

POJO

(Default) Only out-of-band headers are available for filtering, because the in-band headers have already been processed and removed from the list by CXF. The in-band headers are incorporated into the **MessageContentList** in POJO mode. If you require access to headers in POJO mode, you have the option of implementing a custom CXF interceptor or JAX-WS handler.

PAYLOAD

In this mode, both in-band and out-of-band headers are available for filtering.

MESSAGE

Not applicable. (In this mode, the message remains in a raw format and the headers are not processed at all.)

Default filter

The default filter is of type, **SoapMessageHeaderFilter**, which removes only the SOAP headers that the SOAP specification expects an intermediate Web service to consume. For more details, see the section called “SoapMessageHeaderFilter”.

Overriding the default filter

You can override the default **CxfHeaderFilterStrategy** instance by defining a new **CxfHeaderFilterStrategy** bean and associating it with a CXF endpoint.

Sample relayHeaders configuration

The following example shows how you can use the **relayHeaders** option to create a **CxfHeaderFilterStrategy** bean that blocks all message headers. The CXF endpoints in the route use the **headerFilterStrategy** option to install the filter strategy in the endpoint, where the **headerFilterStrategy** setting has the syntax, **headerFilterStrategy=#*BeanID***.

```
<beans ...>
  ...
  <bean id="dropAllMessageHeadersStrategy"
class="org.apache.camel.component.cxf.common.header.CxfHeaderFilterStrateg
y">
      <!--  Set relayHeaders to false to drop all SOAP headers -->
      <property name="relayHeaders" value="false"/>
  </bean>

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
      <route>
          <from uri="cxf:bean:routerNoRelayEndpoint?
headerFilterStrategy=#dropAllMessageHeadersStrategy"/>
          <to uri="cxf:bean:serviceNoRelayEndpoint?
headerFilterStrategy=#dropAllMessageHeadersStrategy"/>
      </route>
```

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

relayAllMessageHeaders option

The **relayAllMessageHeaders** option is used to propagate *all* SOAP headers, without applying any filtering (any installed filters would be bypassed). In order to enable this feature, you must set *both* **relayHeaders** and **relayAllMessageHeaders** to **true**.

Sample relayAllMessageHeaders configuration

The following example shows how to configure CXF endpoints to propagate *all* SOAP message headers. The **propagateAllMessages** filter strategy sets both **relayHeaders** and **relayAllMessageHeaders** to **true**.

```
<beans ...>
  ...
  <bean id="propagateAllMessages"
class="org.apache.camel.component.cxf.common.header.CxfHeaderFilterStrateg
y">
      <!--  Set both properties to true to propagate *all* SOAP headers --
>
      <property name="relayHeaders" value="true"/>
      <property name="relayAllMessageHeaders" value="true"/>
  </bean>

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
      <route>
          <from uri="cxf:bean:routerNoRelayEndpoint?
headerFilterStrategy=#propagateAllMessages"/>
          <to uri="cxf:bean:serviceNoRelayEndpoint?
headerFilterStrategy=#propagateAllMessages"/>
      </route>
  </camelContext>
  ...
</beans>
```

9.2. HEADER FILTERING

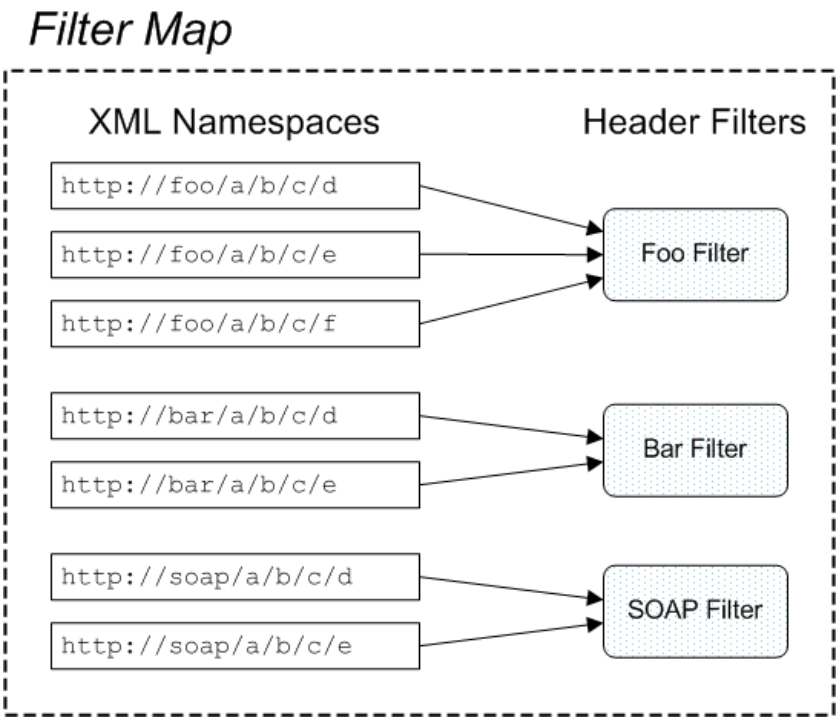
Overview

You can optionally install multiple headers in a **CxfHeaderFilterStrategy** instance. The filtering mechanism then uses the header's XML namespace to lookup a particular filter, which it then applies to the header.

Filter map

Figure 9.1, “Filter Map” shows an overview of the filter map that is contained within a **CxfHeaderFilterStrategy** instance. For each filter that you install in **CxfHeaderFilterStrategy**, corresponding entries are made in the filter map, where one or more XML schema namespaces are associated with each filter.

Figure 9.1. Filter Map



Filter behavior

When a header is filtered, the filter mechanism peeks at the header to discover the header's XML namespace. The filter then looks up the XML namespace in the filter map to find the corresponding filter implementation. This filter is then applied to the header.

PAYLOAD mode

In PAYLOAD mode, both in-band and out-of-band messages pass through the installed filters.

POJO mode

In POJO mode, only out-of-band messages pass through the installed filters. In-band messages bypass the filters and are propagated by default.

9.3. IMPLEMENTING A CUSTOM FILTER

Overview

You can implement your own customer message header filters by implementing the **`MessageHeaderFilter`** Java interface. You must associate a filter with one or more XML schema namespaces (representing the header's namespace) and it is possible to differentiate between request message headers and response message headers.

MessageHeaderFilter interface

The **`MessageHeaderFilter`** interface is defined in the **`org.apache.camel.component.cxf.common.header`** package, as follows:

```
// Java
```

```
package org.apache.camel.component.cxf.common.header;

import java.util.List;

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

public interface MessageHeaderFilter {
    List<String> getActivationNamespaces();

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

Implementing the filter() method

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

`Direction.OUT`

When the **`direction`** parameter equals **`Direction.OUT`**, the filter is being applied to a request either leaving a consumer endpoint or entering a producer endpoint (that is, it applies to a WS request message propagating through a route).

`Direction.IN`

When the **`direction`** parameter equals **`Direction.IN`**, the filter is being applied to a response either leaving a producer endpoint or entering a consumer endpoint (that is, it applies to a WS response message being sent back).

Filtering can be applied by removing elements from the list of headers, **`headers`**. Any headers left in the list are propagated.

Binding filters to XML namespaces

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

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

Identifying the namespace to bind to

Example 9.1, “Sample Binding Namespaces” illustrates how to identify the namespaces to which you can bind a filter. This example shows the WSDL file for a Bank server that exposes SOAP endpoints.

Example 9.1. Sample Binding Namespaces

```
<wsdl:definitions
targetNamespace="http://cxf.apache.org/schemas/cxf/idl/bank"
    xmlns:tns="http://cxf.apache.org/schemas/cxf/idl/bank"
```

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

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

Implementing a custom filter

If you want to implement your own custom filter, define a class that inherits from the **MessageHeaderFilter** interface and implement its methods as described in this section. For example, Example 9.2, “Sample Header Filter Implementation” shows an example of a custom filter, **CustomHeaderFilter**, that binds to the namespace, **http://cxf.apache.org/bindings/custom**, and relays all of the headers that pass through it.

Example 9.2. Sample Header Filter Implementation

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

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

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

public class CustomHeaderFilter implements MessageHeaderFilter {

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

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

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

9.4. INSTALLING FILTERS

Overview

To install message header filters, set the **`messageHeaderFilters`** property of the **`CxfHeaderFilterStrategy`** object. When you initialize this property with a list of message header filters, the header filter strategy combines the specified filters to make a filter map.

The **`messageHeaderFilters`** property is of type, **`List<MessageHeaderFilter>`**.

Installing filters in XML

The following example shows how to create a **`CxfHeaderFilterStrategy`** instance, specifying a customized list of header filters in the **`messageHeaderFilters`** property. There are two header filters in this example: **`SoapMessageHeaderFilter`** and **`CustomHeaderFilter`**.

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

            <!--  Add custom filter here -->
            <bean
class="org.apache.camel.component.cxf.soap.headers.CustomHeaderFilter"/>
        </list>
    </property>
    <!-- The 'relayHeaders' property is 'true' by default -->
</bean>
```

SoapMessageHeaderFilter

The first header filter in the preceding example is the **`SoapMessageHeaderFilter`** filter, which is the default header filter. This filter is designed to filter standard SOAP headers and is bound to the following XML namespaces:

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

This filter peeks at the header element, in order to decide whether or not to block a particular header. If the **`soap:actor`** attribute (SOAP 1.1) or the **`soap:role`** attribute (SOAP 1.2) is present and has the value *next*, the header is removed from the message. Otherwise, the header is propagated.

Namespace clashes

Normally, each namespace should be bound to just a single header filter. If a namespace is bound to more than one header filter, this normally causes an error. It is possible, however, to override this policy

by setting the **`allowFilterNamespaceClash`** property to **`true`** in the **`CxfHeaderFilterStrategy`** instance. When this policy is set to **`true`**, the nearest to last filter is selected, in the event of a namespace clash.