SOA™ software

Policy Manager 7.0 Message Handler Programming Guide

Trademarks

Copyright

Table of Contents

Preface

WHAT'S IN THIS GUIDE?

The *Policy Manager 7.0 Message Handler Programming Guide* provides information about the SOA Container Message Handler Framework. It describes the architecture of the framework, the API of the framework, and how to deploy extensions to the framework.

It includes the following chapters:

- Chapter 1, "Message Handler Framework Architecture," describes the architecture of the Message Handler Framework.
- Chapter 2, "Message Handler Framework API," provides an overview of the classes that make up the Message Handler Framework API.
- Chapter 3, "Message Handler Deployment," describes how Message Handlers are deployed to the Message Handler Framework.
- Chapter 4, "Developing a Message Handler," describes the process for developing and deploying a Message Handler including source code examples.
- JavaDoc API documentation describing the Message Handler Framework API is available in the `\docs\apidocs` folder of your SOA Software Platform release directory.

OTHER DOCUMENTATION

To effectively use this guide, you should have access to and a working knowledge of the concepts outlined in the following Policy Manager product documentation:

- SOA Software Platform 7.0 Installation Guide
- Policy Manager 7.0 Online Help

CUSTOMER SUPPORT

SOA Software offers a variety of support services to our customers. The following options are available:

Support Options:	
Email (direct)	support@soa.com
Phone	1-866 SOA-9876 (1-866-762-9876)
Email (Web)	The "Support" section of the SOA Software website (www.soa.com) provides an option for emailing product related inquiries to our support team.
Documentation Updates	Updates to product documentation are issued periodically and are available by submitting an email request to support@soa.com.

Chapter 1: Message Handler Framework Architecture

OVERVIEW

The SOA Software SOA Container utilizes an extensible Message Handler Framework for executing business logic on messages. The framework is similar in many respects to the JAX-RPC Message Handler Framework, however the SOA Software framework is not limited to SOAP messages only.

The Message Handler framework provides a set of interfaces that can be implemented by developers that would like to extend the base capabilities of a SOA Container. Many of the base capabilities in the container are also implemented using the same framework.

The handler framework is used to process incoming and outgoing messages of web services. The processing is typically constrained to binding specific logic, header processing, and minor transformations. It is not intended to provide orchestration, content based routing, or major transformations. Those capabilities should be pursued through the Virtual Service Orchestration Framework.

The Message Handler Framework is utilized by individual features such as the Delegate, Agent, and Network Director features. Due to the different purposes of the features, they each make use of the framework in different ways. For example, the Agent only takes on the role of a provider and processes incoming messages and therefore the framework is only used to handle incoming messages. Conversely, the Delegate feature acts as a consumer and only processes outgoing messages. The Network Director acts as a provider and consumer of services and therefore the framework is used for processing both incoming and outgoing messages

In all the features, the Message Handler Framework is made up of the same fundamental components, Message Handlers, Handler Factories, and Handler Chains. These components are described in the next section.

MESSAGE HANDLERS

A MessageHandler is a Java class that is given a message from a message exchange to perform some business logic. The handler may or may not make changes to the message. For example, it may process a SOAP WS-Security header and remove it from the message, or it may store some metrics about the message without making any modifications. The MessageHandler can provide feedback that would dictate future processing of the message exchange, such as returning a fault to the consumer.

HANDLER FACTORIES

A MessageHandler is constructed by a HandlerFactory. A HandlerFactory is presented context by the framework that will be used to create the handler. As illustrated in upcoming sections, the context will be different for certain factories based on how they are deployed in the framework. Some examples of the context provided to factories are WSDL descriptions of a service (service specific), WSDL descriptions of a specific operation (operation specific), or an Effective WS-Policy (policy enforcement/implementation).

HANDLER CHAINS

A HandlerChain is a list of MessageHandlers that are invoked in order, each being given the same message as context. The context supports the addition and retrieval of properties. This allows MessageHandlers to associate data with a context without changing the message that a subsequent handler in the chain can use in its business logic.

One HandlerChain is used to process both request (IN) and response (OUT) messages in TWO-WAY message exchanges. The order of handler invocation for OUT messages is configurable. Handlers can be called in reverse order for OUT messages.

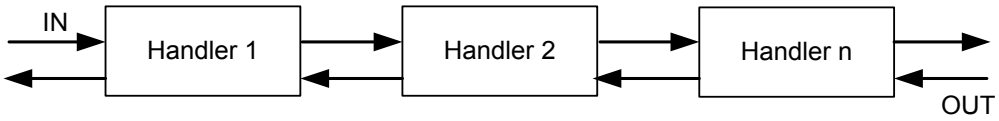

Figure 1-1

Or Handlers can be called in the same order as was followed for IN messages.

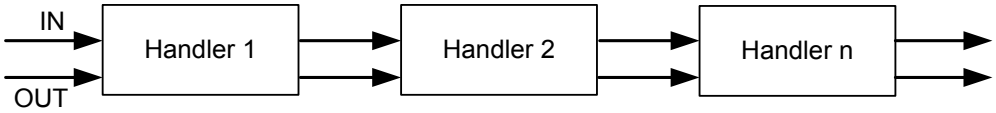

Figure 1-2

A handler can dictate the flow of the HandlerChain by either reporting to the chain that a fault has been encountered or that the handler believes all processing should stop and the exchange should be completed. For example, if a handler encounters an error processing the SOAP WS-Security header of a message, it can relay the error to the HandlerChain which will then relay the error to the framework which will return a fault.

THE FRAMEWORK IN THE AGENT FEATURE

In the Agent feature the Message Handler Framework process all messages in exchanges between an external consumer and a local service implementation.

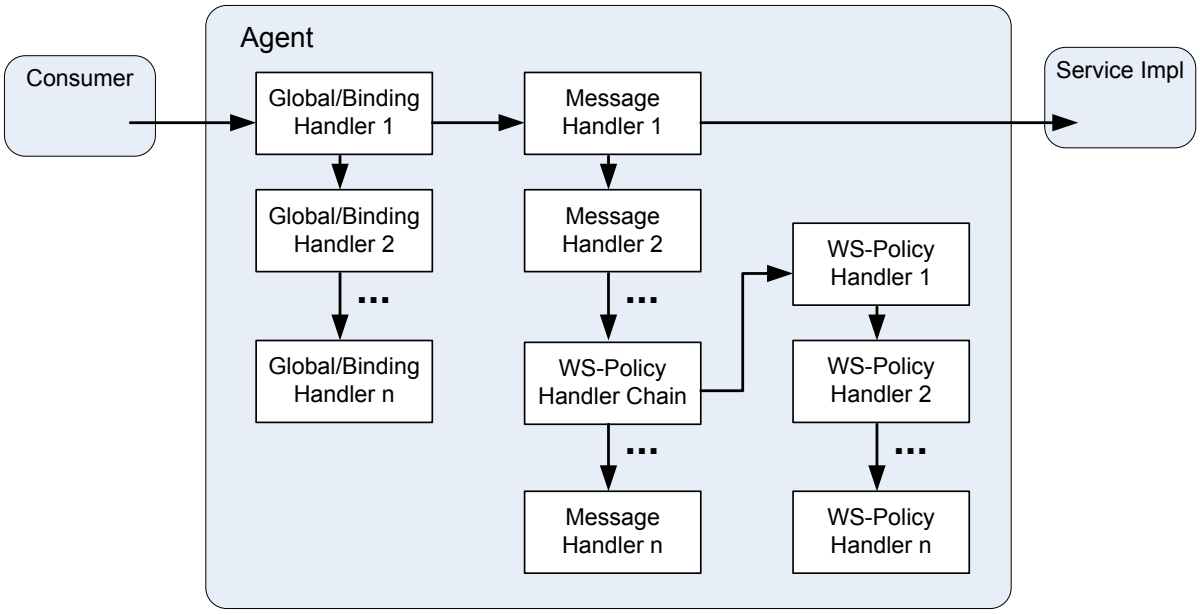

Figure 1-3

The handlers deployed in the framework are organized into three separate groups, *global/binding handlers*, *message specific handlers*, and *WS-Policy handlers*.

The Global/binding handlers are deployed as members of one HandlerChain. There is one chain configured to handle all messages sent through the framework regardless of the service or operation the message invokes. The handlers can be deployed to apply only to messages sent through one binding (i.e., SOAP) or globally to messages sent through all bindings.

Message specific handlers are called after the global/binding handlers. There is one HandlerChain created for each operation/message (IN, OUT, FAULT) of each service managed by the Agent. In cases where an operation has more than one fault message defined, a HandlerChain will be created for each fault. This allows each HandlerFactory to configure each MessageHandler instance with exactly what needs to be done for the targeted operation message, eliminating a lot of potential decisions that could slow down the overall message processing.

WS-Policy handlers is the third group of handlers which are a subset of the message specific handlers with the specific purpose to enforce WS-Policies. These handlers are created by factories that are given the Effective WS-Policy for the given message in addition to the other operation specific context. These handlers behave in the same fashion as other MessageHandlers once configured in a HandlerChain. What sets them apart is the policy context they receive at creation.

The WS-Policy handlers do not necessarily execute after the other message specific handlers. The WS-Policy HandlerChain acts as just one handler in the operation specific HandlerChain. The order in which it is invoked is defined during deployment time and will be discussed in a subsequent section.

THE FRAMEWORK IN THE DELEGATE FEATURE

In the Delegate feature the Message Handler Framework process all messages in exchanges between an internal consumer and a remote service provider.

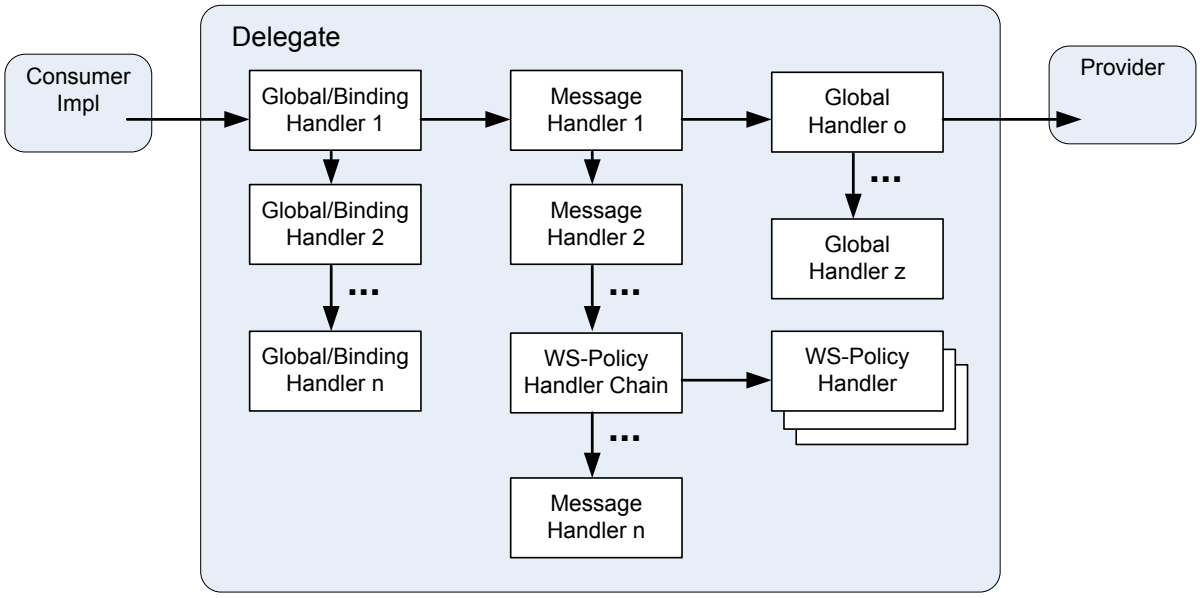

Figure 1-4

Similar to the Agent, the handlers deployed in the framework are organized into three separate groups, global/binding handlers, message specific handlers, and WS-Policy handlers. The only difference in the Delegate case is that the WS-Policy handlers will implement policies rather than enforce them.

THE FRAMEWORK IN THE NETWORK DIRECTOR FEATURE

As mentioned previously, the Network Director acts as an intermediary and therefore both a provider and consumer. It incorporates the use of the framework similar to both the Agent and the Delegate, but is more complex because of its multiple binding and mediation support.

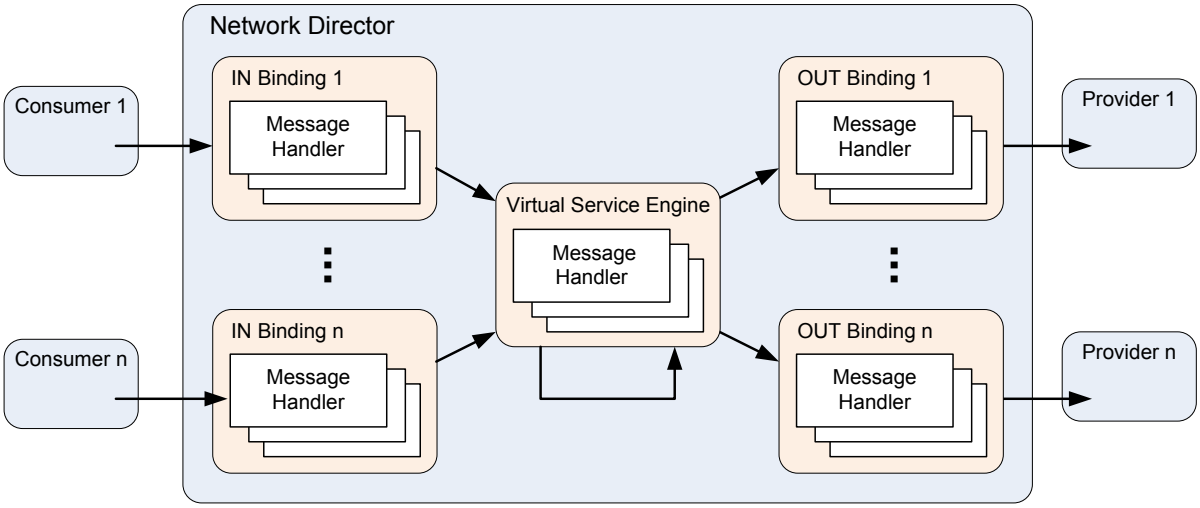

Figure 1-5

The Network Director will support any number of bindings for both incoming and outgoing message exchanges. Based on routing dictated by the Virtual Service engine, messages received on one binding may or may not be forwarded on to the downstream service using the same type of binding.

Every binding implementation is different. Third parties and customers themselves can implement their own bindings. SOA Software developed bindings all incorporate the Message Handler Framework in a consistent fashion. The handlers created and invoked within the bindings may be different based on binding type, but the frameworks will still share a similar organization.

For IN bindings, the framework is organized in the same fashion as the Agent. Each binding only deploys binding handlers that are specific to the matching type of binding in addition to all global handlers. There is one organizational difference however in the Network Director. In Network Director the WS-Policy handlers are divided between the IN bindings and the Virtual Service Engine. The Virtual Service Engine deploys all WS-Policy handlers for policies that are attached to the abstract WSDL components of a Service (PortType, PortType Operation, and Service). The IN bindings deploy the WS-Policy handlers that are specific to the concrete WSDL components of a service (Binding, Binding Operation, and Port). This enables virtual services to invoke other locally deployed virtual services while still having policies enforced.

For OUT bindings, the framework is organized in the same fashion as the Delegate with the additional binding specific group described for IN bindings.

Chapter 2: Message Handler Framework API

The core of the Message Handler Framework is composed of a small number of interfaces and classes and includes an API for creating handlers. This section provides a brief description of these interfaces and classes. A detailed description of the API can be found with the API documentation installed with the Policy Manager product. A UML diagram of the primary interfaces and classes of the framework is illustrated below.

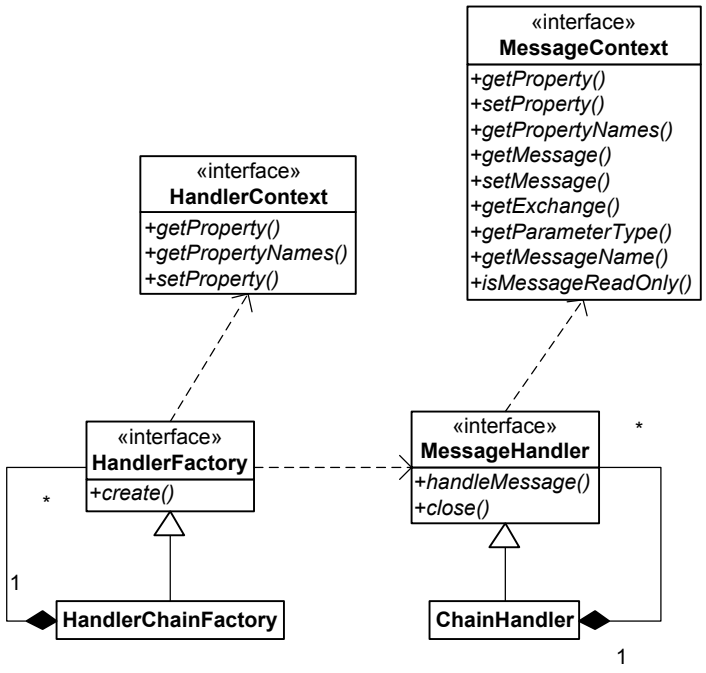

Figure 2-1

- The **MessageHandler** interface is implemented to provide the business logic to perform on a message. The framework will invoke the handleMessage() method with a MessageContext object. The same method is invoked for IN, OUT, and FAULT messages. Depending on deployment, the same instance of a MessageHandler may or may not be called. The close() method is invoked by the framework after it has completed all handler invocations for a given message. This allows handlers to de-allocate resources and perform cleanup outside the processing time of the message itself.
- The **MessageContext** interface gives a MessageHandler access to all the information the framework knows about the message and exchange being processed. The message itself is provided in the context as well as properties about the message that may have been created from other handlers. The exchange is also available in the context. If processing an OUT

message, the IN message will be available from the exchange. Also, properties that are associated with the exchange as a whole can also be created and retrieved.

- The **HandlerFactory** interface is implemented to create instances of a MessageHandler implementation. It is the HandlerFactory implementations that are deployed to the framework. They in turn create the MessageHandler implementations when requested based on how they are deployed. The HandlerFactory is given a HandlerContext that provides contextual information that the factory can use to configure the instance of the MessageHandler.
- The **HandlerContext** interface gives a HandlerFactory access to deployment information. For example, if the HandlerFactory is deployed in the global group, it is given very minimal information. However if the HandlerFactory is deployed in the message specific group, it is given access to the WSDL that describes the message that will need to be processed.
- The **ChainHandler** is a MessageHandler implementation that aggregates a list of MessageHandlers and invokes them in order. An author of a MessageHandler may find it useful to aggregate other MessageHandlers in which case the ChainHandler would be ideal.
- The **HandlerChainFactory** is a HandlerFactory implementation that creates a ChainHandler. It aggregates other HandlerFactory implementations that will create the MessageHandler implementations that the ChainHandler will aggregate.

Chapter 3: Message Handler Deployment

The Network Director, Agent, and Delegate use the OSGi (Open Services Gateway initiative) framework for deploying features and extensions. The Message Handler Framework in each of these environments will dynamically construct their chain of handlers by discovering OSGi services that are published by OSGi Bundles.

The Message Handler Framework registers with the OSGi framework for services that implement the HandlerFactory interface. It organizes the HandlerFactory services into groups as described earlier through the use of attributes that the HandlerFactory services can use to describe themselves. The following are the attributes the Message Handler Framework will use to group services.

Name	Description
name	Names the handler. Can be used by another handler if it needs to state a direct dependency on this handler (see before and after attributes).
scope[1]	Indicates which organizational group the handlers from the factory should be placed in. The values are: • all – Deploy a MessageHandler instance for all messages. This is a "global" handler. • binding – Deploy a MessageHandler instance for a specific type of binding (see binding attribute for which type). • binding.operation – Deploy a MessageHandler instance for each operation message. This is a "message" handler.
binding	Indicates which binding (if the scope attribute value is binding) the handlers from the factory should be deployed for.
role	Indicates whether the handlers from the factory should be used for IN messages (Agent and IN bindings) or OUT messages (Delegate and OUT bindings). The values are: • consumer – Used for OUT messages • provider – Used for IN messages
before	Specifies an ordering requirement or dependency within the

[1] The scope attribute will have different values for WS-Policy Handlers which are not detailed in this document.

Name	Description
	group of handlers it is deployed to. The value is either the name of another handler or the wildcard (*). If '*' is specified, the handler must be placed **before** all other handlers in the group. If multiple handlers have the same value, the framework will order them in the order the OSGi framework discovers them.
after	Specifies an ordering requirement or dependency within the group of handlers it is deployed to. The value is either the name of another handler or the wildcard (*). If ;'*' is specified, the handler must be placed **after** all other handlers in the group. If multiple handlers have the same value, the framework will order them in the order the OSGi framework discovers them.

The following is an example of how handlers can be defined as OSGi services and what the resulting invocation order will be. WS-Policy handlers are not included as they are described in a separate technical note.

Defining the following services in an Agent:

Handler1
Name: Handler1
Scope: all
Role: provider

Handler2
Name: Handler2
Scope: all
Role: provider
Before: *

Handler3
Name: Handler3
Scope: binding
Binding: soap
Role: provider

Handler4
Name: Handler4
Scope: binding.operation
Binding: soap
Role: provider
After: Handler5

Handler5
Name: Handler5
Scope: binding.operation

Binding: soap
Role: provider

, will result in the following deployment.

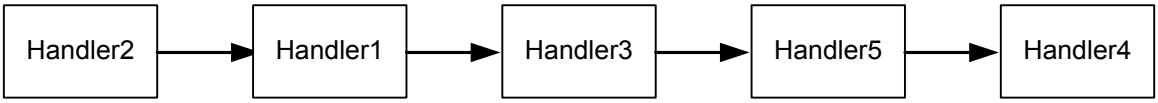

Figure 3-1

Handler1, Handler2, and Handler3 are in the same global/binding group and are deployed first. Handler2 is given the first position in the invocation order because it specified a "before" attribute of "*." Handler1 or Handler3 could have been second since there were no ordering constraints on either one, but in this example Handler1 will be second. Handler4 and Handler5 are in the second message specific group. Handler5 will be deployed before Handler4 though because of Handler4's "after" attribute which referred directly to Handler5.

In the Delegate these same example services would be defined with the same attributes except for the "role," which would be have the value of "consumer." The deployment order would be the same.

Chapter 4: Developing a Message Handler

This section will describe the steps necessary to develop and deploy a MessageHandler to a SOA Container. The sample artifacts described are available in the samples directory installed with the product.

In the example, a MessageHandler will be developed that will simply log the contents of the IN and OUT messages of a message exchange. The MessageHandler will be a "global" handler and can therefore be deployed to any binding.

SOURCE CODE

The source code of the LoggingHandler (a MessageHandler implementation) is as follows:

```
01) package com.soa.examples.handler.logging;
02)
03) import java.io.StringWriter;
04)
05) import javax.xml.transform.Source;
06) import javax.xml.transform.Transformer;
07) import javax.xml.transform.TransformerFactory;
08) import javax.xml.transform.stream.StreamResult;
09)
10) import com.digev.fw.log.Log;
11) import com.digev.fw.log.LogLevel;
12) import com.soa.message.handler.MessageContext;
13) import com.soa.message.handler.MessageFaultException;
14) import com.soa.message.handler.MessageHandler;
15)
16) /**
17)  * MessageHandler implementation that logs the content of a message.
18)  */
19) public class LoggingHandler implements MessageHandler {
20)
21)     private static final Log log = Log.getLog(LoggingHandler.class);
22)
23)     public void close(MessageContext context) {
24)         // no cleanup necessary
25)     }
26)
27)     /* Logs the content of the message */
28)     public boolean handleMessage(MessageContext context)
29)             throws MessageFaultException {
30)         try {
31)             // get the message from the context
32)             Source msgContent = context.getMessage().getContent();
33)             // log the message content as an informative message
34)             log.logText(msgToString(msgContent), LogLevel.INFO);
35)             return true; // continue handler processing
36)         } catch (Exception e) {
37)             throw new MessageFaultException(e);
38)         }
```

```
39)     }
40)
41)     /* Transforms the Source content of a message to a String for
42)      * logging.
43)      */
44)     private String msgToString(Source msg) throws Exception {
45)         Transformer xformer =
46)                 TransformerFactory.newInstance().newTransformer();
47)         StringWriter writer = new StringWriter();
48)         StreamResult result = new StreamResult(writer);
49)         xformer.transform(msg, result);
50)         return writer.toString();
51)     }
52) }
```

Figure 4-1

On line 32, the message content is retrieved from the message context as a java.xml.transform.Source object. That content is converted to a String for logging using the msgToString() method on lines 44 – 51. The SOA Software Logging Framework is used to log the content to the SOA Container's log file on line 34.

If any exception is thrown during the processing a fault will be returned to the consumer and all remaining handlers in the chain will not be invoked. This is done by creating a MessageFaultException in the catch block on lines 36 – 38. The MessageFaultException supports defining fault context information such as a code and description. In this example, nothing is provided except the causing exception which will result in a default fault code being used.

If the logic completes without exception the handler returns true on line 35. This indicates to the framework that it should continue invoking handlers in the handler chain.

The close() method on lines 23 – 25 perform no function in this example. If the handler were to have allocated resources that should be cleaned up only after the entire handler chain had finished it's processing, it would have been done here.

The source code of the LoggingHandlerFactory (a HandlerFactory implementation) is as follows:

```
01) package com.soa.examples.handler.logging;
02)
03) import com.digev.fw.exception.GException;
04) import com.soa.message.handler.HandlerContext;
05) import com.soa.message.handler.HandlerFactory;
06) import com.soa.message.handler.HandlerRole;
07) import com.soa.message.handler.MessageHandler;
08)
09) /**
10)  * Creates a LoggingHandler.
11)  */
12) public class LoggingHandlerFactory implements HandlerFactory {
13)
14)     public MessageHandler create(HandlerContext context,
15)                                  HandlerRole    role)
16)             throws GException {
17)         return new LoggingHandler();
18)     }
19) }
```

Figure 4-2

Since there is no configuration needed, the only step for the factory is to construct a new LoggingHandler with the default constructor.

BUNDLE

The LoggingHandler and LoggingHandlerFactory classes need to be packaged in an OSGi Bundle so that they can be deployed to the SOA Container. The LoggingHandlerFactory needs to be published as an OSGi service so that the Message Handler Framework can load it. In this example, Spring DM (Distributed Modules for OSGi) is used to construct and publish the LoggingHandlerFactory OSGi service. Spring DM is not a requirement but is used here for simplicity.

```
01) <?xml version="1.0" encoding="UTF-8"?>
02)
03) <!--
04)   Spring definition for the logging handler OSGi integration.
05) -->
06) <beans xmlns="http://www.springframework.org/schema/beans"
07)        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
08)        xmlns:osgi="http://www.springframework.org/schema/osgi"
09)        xsi:schemaLocation="http://www.springframework.org/schema/beans
10)
 http://www.springframework.org/schema/beans/spring-beans.xsd
11)
 http://www.springframework.org/schema/osgi
12)
 http://www.springframework.org/schema/osgi/spring-osgi.xsd">
13)
14)      <osgi:service interface="com.soa.message.handler.HandlerFactory">
15)              <osgi:service-properties>
16)                      <entry key="name" value="examples.logging.factory"/>
17)                      <entry key="scope" value="all"/>
18)                      <entry key="role" value="provider"/>
19)              </osgi:service-properties>
20)              <bean
class="com.soa.examples.handler.logging.LoggingHandlerFactory"/>
21)      </osgi:service>
22)
23) </beans>
```

Figure 4-3

The OSGi service is defined on lines 14 – 21. The service is published as an implementation of the HandlerFactory interface using the "interface" attribute. The service has three attributes defined on lines 15 – 19. The scope is "all" which indicates to the framework to deploy this handler as a "global" handler to all bindings of the container. The role is "provider" which indicates to the framework that the handler will process incoming messages to the container. This would not work in the Delegate as the Delegate acts only as a "consumer."

The construction of the LoggingHandlerFactory is specified on line 20. Since only the default constructor is needed to create the factory, there is nothing more than the identification of the class to instantiate required here.

An OSGi Bundle must have a Manifest to define it's dependencies. The following is the Manifest for this example.

```
01) Manifest-Version: 1.0
02) Bundle-ManifestVersion: 2
03) Bundle-Name: Logging MessageHandler Example
04) Bundle-SymbolicName: com.soa.examples.handler.logging
05) Bundle-Version: 1.0.0
06) Bundle-Vendor: SOA Software
07) Import-Package: com.digev.fw.exception;version="6.0.0",
08)  com.digev.fw.log;version="6.0.0",
09)  com.soa.message;version="6.0.0",
10)  com.soa.message.handler;version="6.0.0",
11)  javax.xml.transform,
12)  javax.xml.transform.stream
```

Figure 4-4

Lines 01 – 06 hold general information about the Bundle. Lines 07 – 12 hold the package dependencies for the Bundle. All packages not defined within the bundle that are imported by code in the Bundle must be listed here. The only exceptions to this are packages that are in the global classpath of the SOA Container such as the Java JRE and Spring packages. A full list of the globally available packages is listed below.

DEPLOYMENT

An SOA Software SOA Container will have a folder on the file system with a name that matches the key of the container seen in the Policy Manager console. Under that folder is a sub-folder named "deploy." Bundles that provide extensions to the container, such as additional message handlers, will be placed in the "deploy" folder. Upon restart of the container, the services published within any Bundles in the "deploy" folder will be imported into the container and all published handler factories will be deployed by the Message Handler Framework.