SOA™ software

Policy Manager 7.x Policy Handler Programming Guide

Trademarks

Copyright

Table of Contents

Preface

WHAT'S IN THIS GUIDE?

The *Policy Manager 7.x Policy Handler Programming Guide* provides information about the SOA Container Policy 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, "Policy Handler Framework Architecture," describes the architecture of the Policy Handler Framework.
- Chapter 2, "Policy Handler Framework API," provides an overview of the classes that make up the Policy Handler Framework API.
- Chapter 3, "Policy Handler Deployment," describes how Policy Handlers are deployed to the Policy Handler Framework.
- Chapter 4, "Developing a Policy Handler," describes the process for developing and deploying a Policy 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 or by visiting http://docs.soa.com/ag/assets/apiDocs_pm72.

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.x Installation Guide
- Policy Manager 7.x Online Help
- Policy Manager 7.x Message Handler Programming Guide

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: Policy Handler Framework Architecture

OVERVIEW

The SOA Software SOA Container utilizes an extensible Policy Handler Framework for implementing and enforcing policies on messages. The Policy Handler Framework is an extension of the Message Handler Framework (see the Policy Manager 7.x Message Handler Programming Guide) specialized for runtime policies, both operational and quality of service.

The Policy Handler framework provides a set of interfaces in addition to those provided by the Message Handler Framework that can be implemented by developers who would like to extend the base policy capabilities of a SOA Container. The base policy capabilities in the product are also implemented using the same framework.

The policy handler framework is used to process incoming and outgoing messages of web services. This processing is typically constrained to binding specific logic, header processing, security decisions, 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 Policy Handler Framework is utilized by individual features such as the Delegate, Agent, and Network Director. 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 message exchanges. Therefore the framework is only used to handle incoming requests and their responses. Conversely, the Delegate feature acts as a consumer and only processes outgoing message exchanges. The Network Director acts as a provider and consumer of services and therefore the framework is used for processing both incoming and outgoing message exchanges.

The processing performed by the policy handler framework is dictated by policies attached to services in Policy Manager. Enforcement is the act of ensuring a policy is met by a message. Implementation is the act of altering a message so that it conforms to a policy. When the policy framework governs a service that is receiving messages, the framework enforces the policies attached the service. It must also implement those same policies on messages that are returned to the client.

For example, service A has a security policy attached to it that dictates the request and response messages must be signed. The framework will enforce that policy on the request message by verifying a signature is present on the request message and that it is valid. The framework will then implement the policy on the response message by signing it before it is returned to the client.

When the framework is used on the consumer side of a message exchange it implements the policies attached to the target service on the messages sent to the target and enforces the same policies on the messages returned by the target. In the case of the Network Director feature the policy framework is invoked twice, once for the exchange between the client and the virtual service and once for the exchange between the virtual service and the target service.

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

POLICIES

Policies in Policy Manager are modeled according to the [WS-Policy] specification. The [WS-Policy] specification defines a policy as a set of assertions that can be grouped together in a conditional fashion using XML. An assertion is any XML element that represents enforceable or implementable rules.

Both the [WS-Policy] specification and Policy Manager support the notion of assertions themselves having their own policies so that a nesting such as policy -> assertion -> policy -> assertion may be possible. The WS-Policy specification supports the notion of a policy containing choices of assertions at any level. Policy Manager only support choices within policies that are contained within assertions, not directly within the root policy itself.

The specification is flexible about how policies can be constructed but it does provide a single normal form that all policies can be converted to. That normal form is how Policy Manager represents all policies. The following is an example of a policy in Policy Manager

```
01) <wsp:Policy Name="My Policy">
02)   <wsp:ExactlyOne>
03)     <wsp:All>
04)       <MyAssertion>
05)         <wsp:Policy>
06)           <wsp:ExactlyOne>
07)             <wsp:All>
08)               <MyChoice1/>
09)             </wsp:All>
10)             <wsp:All>
11)               <MyChoice2/>
12)             </wsp:All>
13)           </wsp:ExactlyOne>
14)         </wsp:Policy>
15)       </MyAssertion>
16)     </wsp:All>
17)   </wsp:ExactlyOne>
18) </wsp:Policy>
```

Lines 01 – 18 represent a single policy named “My Policy.” The policy author has supplied an assertion named MyAssertion on lines 04 – 15. MyAssertion has provided two choices, MyChoice1 on line 8 and MyChoice2 on line 11. The use of the ExactlyOne element on line 6 delineates the choices.

Multiple policies can be attached to services and organizations in Policy Manager. These policies can be attached to different levels of the organization tree and different levels of the service definition. All the policies that apply to a given request or response message must be collected and combined so that they can be properly enforced or implemented. All these policies are combined into what is called an "effective" policy, or the complete set of assertions that apply to a given message. There will be an effective policy for each message (IN, OUT, FAULT) of each operation of each service being governed as described in a WSDL document. For more information about policy attachments (scopes) and "effective" policies please consult the WS-Policy Attachment specification.

To illustrate, a policy with MyAssertion1 is attached to a service in Policy Manager. Another policy with MyAssertion2 is attached to an operation of that service in Policy Manager. The effective policy for the operation would appear as below:

```
01) <wsp:Policy Name="My Effective Policy">
02)   <wsp:ExactlyOne>
03)     <wsp:All>
04)       <MyAssertion1/>
05)       <MyAssertion2/>
06)     </wsp:All>
07)   </wsp:ExactlyOne>
08) </wsp:Policy>
```

For more information about policy attachments (scopes) and "effective" policies please consult the [WS-PolicyAttachment] specification.

MARSHALLERS

The Policy Handler Framework will receive policies from Policy Manager in the XML form described previously. It will parse the XML into a Java representation that can then be used by policy handlers to implement and enforce. The framework has a Java API that it parses the policy constructs (Policy, ExactlyOne, All) into but it doesn't have an understanding of the assertions policy authors write. Instead it delegates the assertion interpretation to domain specific implementations provided by the policy authors.

The policy author provides an Assertion Marshaller that parses the assertion XML into a Java representation. If a policy author does not have a domain specific Java model for their assertion they can rely on built-in facilities to marshal the assertion into an `org.w3c.dom` (DOM) representation. A policy author can utilize other XML marshaling frameworks such as the `javax.xml.bind` (JAXB) API within a marshaller as well. Or the author can write proprietary parsing code.

POLICY HANDLERS

A Policy Handler is a Java class that is given a message from a message exchange to either implement or enforce a policy. A Policy Handler is actually just a Message Handler described in the PM 7.x Message Handler Programming Guide. The same Message Handler is used within the Policy Handler Framework. There are no differences between Message Handlers used in

both frameworks. The differences are found in how the handlers are created through the use of factories.

POLICY HANDLER FACTORIES

A Policy (Message) Handler is constructed by a Policy Handler Factory. The framework will call a handler factory with context about the handler that should be created including the "effective" policy and the scope of the handler. Since an effective policy can be different for each message (IN, OUT, FAULT) of each operation of a service, the scope will be the exact message the effective policy is for. In other words, a handler is created for each message of each operation of a service. If an operation has multiple faults defined in its WSDL document the factory will be called for each fault.

There is not a one-to-one relationship of policy to Policy Handler Factory or Policy Handler. Multiple factories and handlers can process a single policy. A single factory or handler can execute business logic based on multiple policies. Because each factory registered with the framework is called with the effective policy it has access to all the assertions present within that effective policy and can read each assertion it understands.

For example, a policy P1 has some conditional rules that are enforced if another policy P2 has been attached to the same message. P2 has its own policy handler factory that interprets it. P1 has to not only interpret P1 but also check for the presence of P2 when building its handler.

HANDLER CHAINS

As described in the PM 7.x Message Handler Programming Guide a Handler Chain is a list of MessageHandlers that are invoked in order, each being given the same message as context. The Policy Handlers created from the Policy Handler Factories will be put in one handler chain. The order of their execution is based on the order in which the factories were called.

THE FRAMEWORK IN THE AGENT FEATURE

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

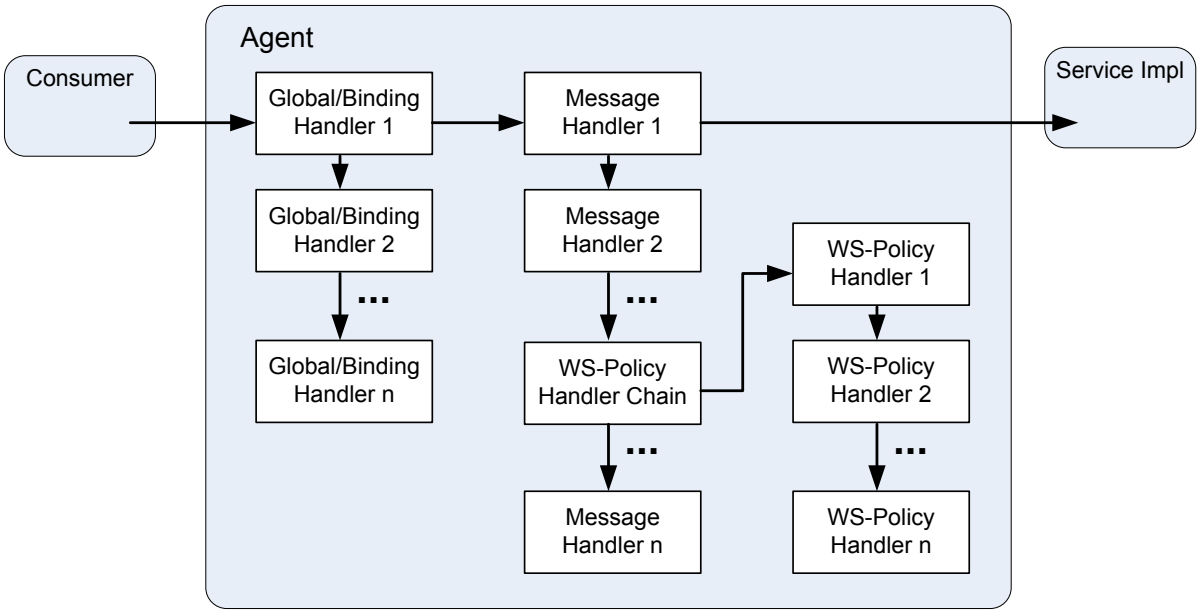


Figure 1-3

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

The Policy Handler Chain holds all the WS-Policy handlers. The chain is executed as one message specific Message Handler.

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.

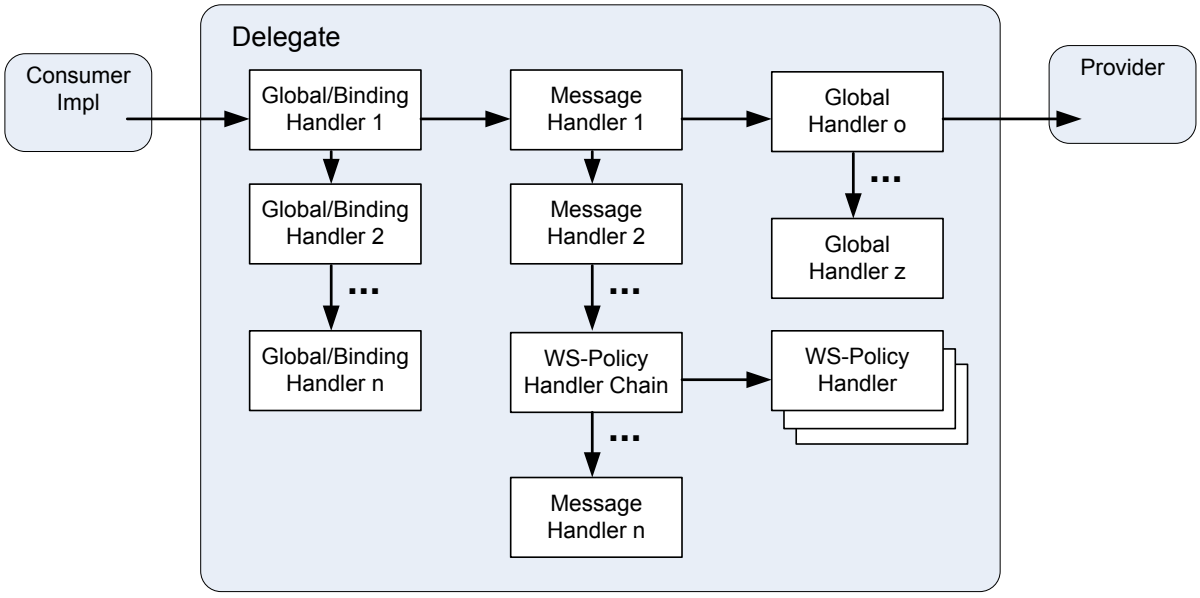


Figure 1-4

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

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.

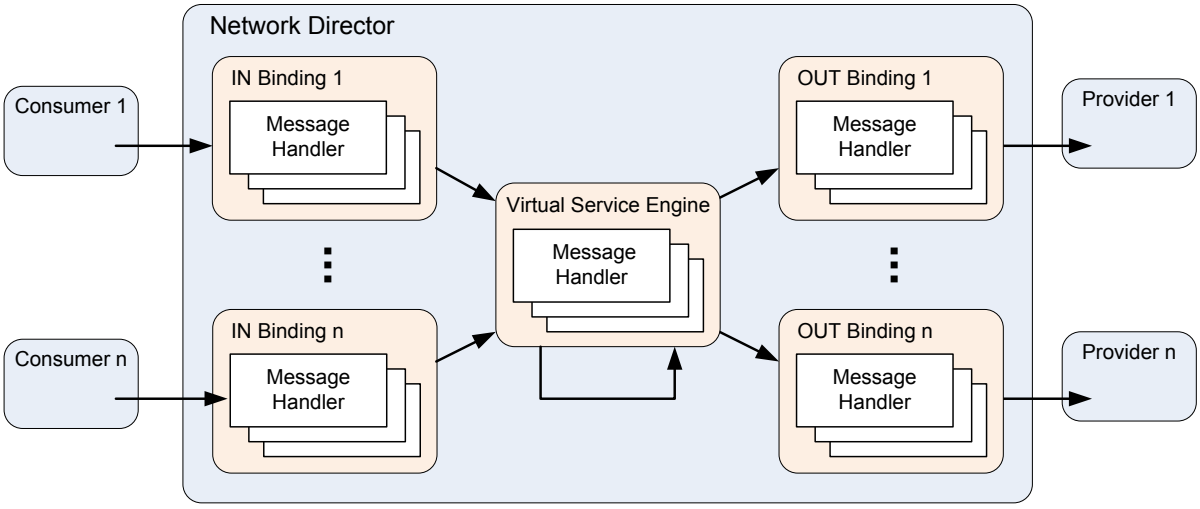


Figure 1-5

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 Policy 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.

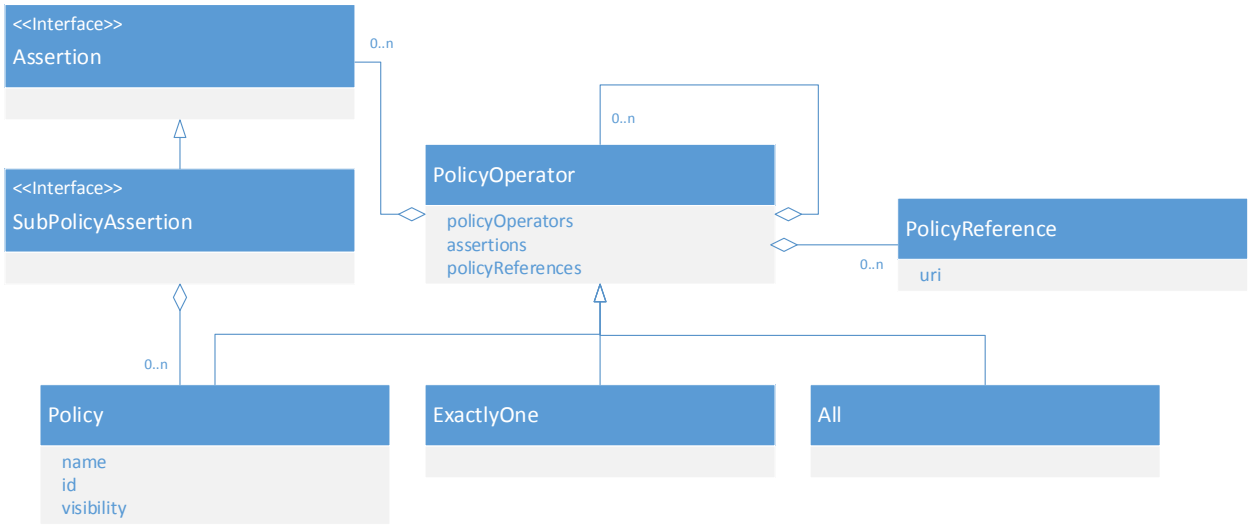
Chapter 2: Policy Handler Framework API

The Policy Handler Framework API is composed of three major groups of classes, Policy API, Message Handler API, and Policy Handler Factory API. The Policy API provides interfaces and classes for defining policies, assertions, and assertion marshalling. The Message Handler API is described in the PM 7.x Message Handler Programming Guide and provides the core interfaces and classes for Message Handlers and processing of message exchanges. The Policy Handler Factory API provides interfaces and classes for creating Message Handlers but within the context of policy enforcement and implementation.

This following sections provide a brief description of these interfaces and classes. A detailed description of the API can be found in the \docs\apidocs folder of your SOA Software Platform release directory or by visiting http://docs.soa.com/ag/assets/apiDocs_pm72.

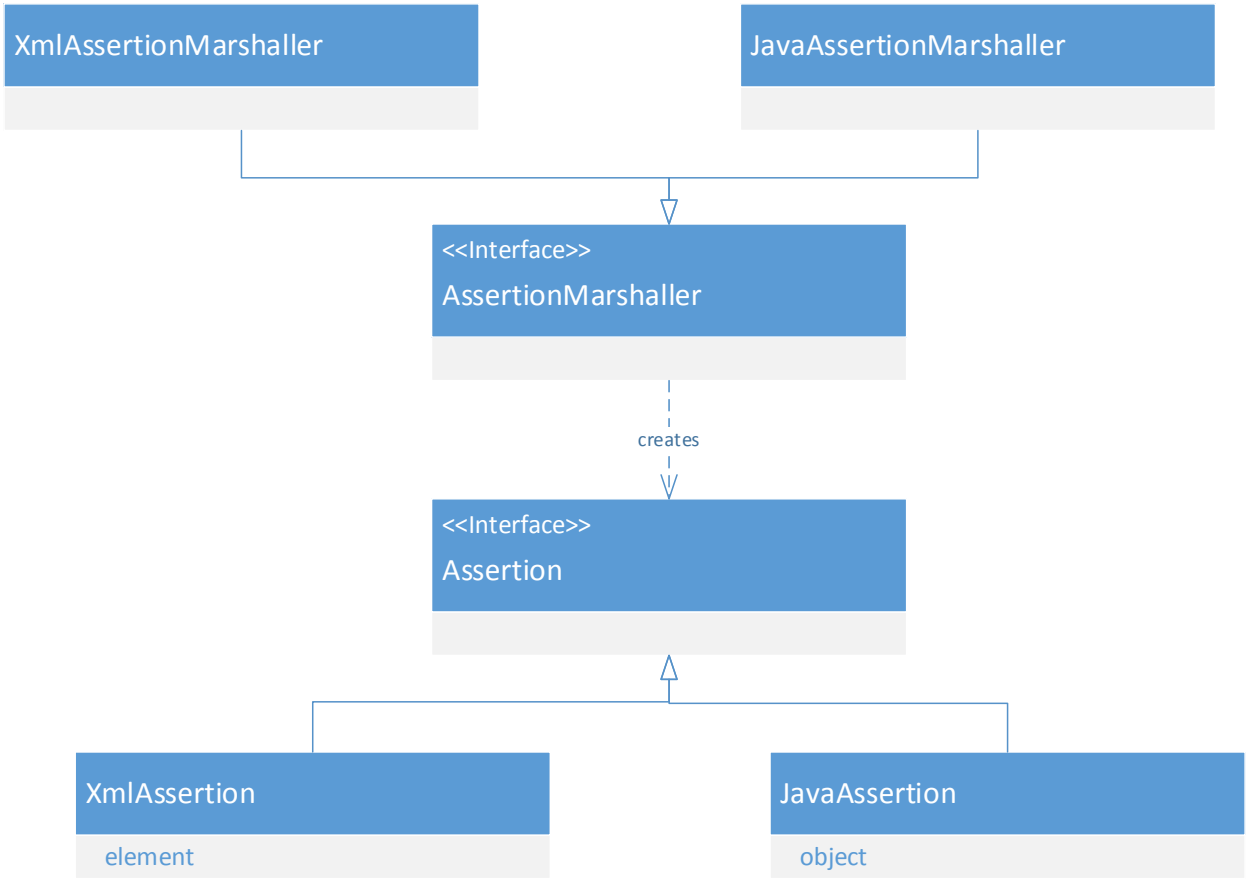
POLICY API

The Policy API is composed of a small number of interfaces and classes that can be used to represent policies and assertions. These are all in the `com.soa.policy.wspolicy` package.



The `Policy`, `ExactlyOne`, and `All` classes are WS-Policy constructs. The `Assertion` interface is what all assertions must implement. The `SubPolicyAssertion` interface is an `Assertion` extension for assertions that have nested policies of their own.

The Policy API also provides an interface for assertion marshalling as well as some pre-existing marshalling implementations.



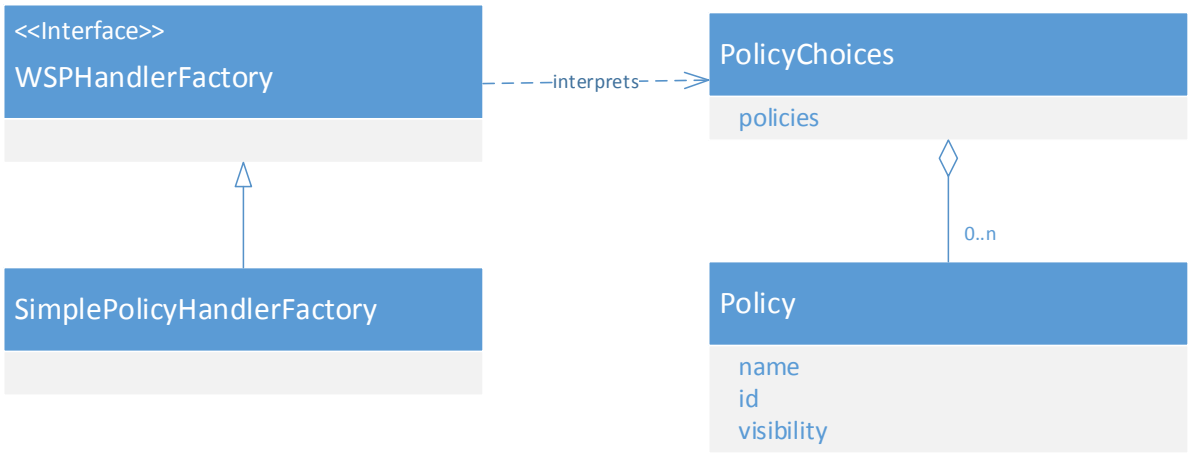
An assertion is represented in the framework with the `Assertion` interface. Assertion is the interface that all domain specific representations must implement. A policy author can implement this interface directly with their own class or they can utilize some of the existing implementations. `XmlAssertion` for example provides a default DOM representation of an assertion. `JavaAssertion` provides an implementation that simply wraps an existing Java object. This is useful when the author wants an assertion class that does not have to implement the Assertion interface, such as when they are using JAXB to model an assertion.

Policy authors instruct the framework how an assertion must be parsed by registering an `AssertionMarshaller`. `AssertionMarshaller` is an interface that an author can implement that will be called by the framework with a DOM element representing an assertion. The author will return an `Assertion` implementation back to the framework that will be passed to policy handlers later. If the author wants to use the `XmlAssertion` they can also use the existing `XmlAssertionMarshaller`. If the author wishes to use JAXB they can use the `JaxbAssertionMarshaller`.

POLICY HANDLER FACTORY API

The Policy Handler Factory API is composed of a small number of interfaces and classes that can be used to provide construction logic for Policy Handlers based on policy assertions

modeled in the Policy API. These can be found in the `com.soa.policy.wspolicy.handler` and `com.soa.policy.wspolicy.handler.ext` packages.



The `WSPHandlerFactory` is the interface all policy handler factories must implement. It is different from a `HandlerFactory` in the Message Handler Framework in that it is given the effective policy as a set of normalized policy choices represented with the `PolicyChoices` class. Currently Policy Manager does not allow policy choices at the root of its effective policy, only within assertions themselves. So, with this limitation in mind the `SimplePolicyHandlerFactory` abstract class is provided for policy authors to extend for their policy handler factories. This class provides subclasses with a single choice as the effective policy which simplifies processing.

Chapter 3: Policy Handler Deployment

The Network Director, Agent, and Delegate use the OSGi (Open Services Gateway initiative) framework for deploying features and extensions. The Policy Handler Framework in each of these environments will dynamically construct their chain of handlers by discovering policy handler factories published as OSGi services by OSGi bundles.

The Policy Handler Framework registers with the OSGi framework for services that implement the `WSPHandlerFactory` interface. It organizes the `WSPHandlerFactory` services into groups as described in the framework/feature sections through the use of attributes that the `WSPHandlerFactory` services can use to describe themselves. The following are the attributes the Policy Handler Framework will use to group `WSPHandlerFactory` services.

Name	Description
name	Names the handler factory. Can be used by another handler factory if it needs to state a direct dependency on this handler factory (see before and after attributes).
scope	Indicates which organizational group the handlers from the factory should be placed in. The values are: • concrete – Deploy a factory instance for a specific binding (see the binding property). • abstract – Deploy a factory instance at the service level so it will create a handler for messages sent/received over any binding.
binding	Indicates which binding (if the scope attribute value is "concrete") the factory should be deployed for.
role	Indicates whether the handlers from the factory should be used for receiving message exchanges (Agent and virtual services) or initiating message exchanges (Delegate and downstream services). The values are: • consumer – Used for initiating exchanges • provider – Used for receiving exchanges
before	Specifies an ordering requirement or dependency within the group of handler factories it is deployed to. The value is either the name of another handler factory or the wildcard (*). If '*' is specified, the factory must be placed **before** all other factories in the group. If multiple factories have the same value, the framework will order them in the order the OSGi framework

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

The following is an example of how policy handler factories can be defined as OSGi services and what the resulting invocation order will be.

Defining the following services in an Agent:

Factory1
Name: Factory1
Scope: abstract
Role: provider

Factory2
Name: Factory2
Scope: abstract
Role: provider
Before: *

Factory3
Name: Factory3
Scope: concrete
Binding: soap
Role: provider

Factory4
Name: Factory4
Scope: concrete
Binding: soap
Role: provider
After: Handler5

Factory5
Name: Factory5
Scope: concrete
Binding: soap
Role: provider

will result in the following deployment.

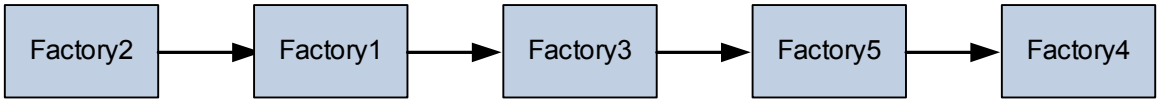


Figure 3-1

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 Policy Handler

This section describes the steps necessary to develop and deploy a Policy Handler to an SOA Container. The sample artifacts described are available in the samples directory installed with the product.

In the example, a policy will be written that will dictate that a transport header be present with the same value as the operation name as defined by the service's WSDL document. Policy handler will be written to both enforce and implement the policy.

Policy Assertion Schema

The assertion used in the example is defined by the following XML schema.

```
01) <?xml version="1.0" encoding="UTF-8"?>
02) <xs:schema
targetNamespace="http://soa.com/products/policymanager/examples/policy/complex"
elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
03)   <xs:element name="Complex">
04)     <xs:complexType>
05)       <xs:sequence>
06)         <xs:element name="HeaderName" type="xs:string"></xs:element>
07)         <xs:element name="Optional" type="xs:boolean"></xs:element>
08)       </xs:sequence>
09)     </xs:complexType>
10)   </xs:element>
11) </xs:schema>
```

The assertion name is "Complex" as defined on line 03. The assertion has two elements, HeaderName on line 06 which identifies the name of the header the operation name must be in, and Optional on line 07 which indicates whether the presence of the header is optional or required.

Source Code

In this example JAXB is going to be used to bind our XML assertion to Java. The generated JAXB java class that represents the Complex assertion is below.

```
01) @XmlAccessorType(XmlAccessType.FIELD)
02) @XmlType(name = "", propOrder = {
03)   "headerName",
04)   "optional"
05) })
06) @XmlRootElement(name = "Complex")
07) public class Complex {
08)
09)   @XmlElement(name = "HeaderName", required = true)
```

```
10)   protected String headerName;
11)   @XmlElement(name = "Optional")
12)   protected boolean optional;
13)
14)   public String getHeaderName() {
15)     return headerName;
16)   }
17)
18)   public void setHeaderName(String value) {
19)     this.headerName = value;
20)   }
21)
22)   public boolean isOptional() {
23)     return optional;
24)   }
25)
26)   public void setOptional(boolean value) {
27)     this.optional = value;
28)   }
29)
30) }
```

When using JAXB a `JavaAssertion` will be what is generated when unmarshalling the policy. The `Complex` class will be contained within the `JavaAssertion`. In this example an optional step is taken where the `JavaAssertion` class is extended to provide methods that mimic the `Complex` class so that clients are unaware of the use of JAXB or the necessity to extract the JAXB object from the `JavaAssertion`. Once again, this is purely optional.

```
01) public class ComplexAssertion extends JavaAssertion {
02)
03)   private static Log log = Log.getLog(ComplexAssertion.class);
04)
05)   private Complex complex = new Complex();
06)
07)   private  Complex getComplex() {
08)     Complex complexPolicy  = null;
09)     try{
10)       if (getObject() instanceof Complex){
11)         complexPolicy = (Complex)getObject();
12)       }
13)       else {
14)         throw new RuntimeException(
15)             "The object " + getObject()+ " is not an Complex");
16)       }
17)     }
18)     catch (Throwable t){
19)       log.error(t);
20)     }
21)     return complexPolicy;
22)   }
23)
24)   private Complex createComplex(){
25)     if (super.getObject() == null){
26)       try{
27)         Complex complexPolicy = new Complex();
28)         complexPolicy.setHeaderName("");
29)         super.setObject(complexPolicy);
30)       }
31)       catch (Throwable t){
32)         log.error(t);
33)       }
```

```
34)     }
35)     if(!(super.getObject() instanceof Complex))  {
36)       throw new RuntimeException(
37)           "The object " + getObject()+ " is not an Complex");
38)     }
39)     return (Complex)super.getObject();
40)   }
41)
42)   public String getHeaderName() {
43)     return getComplex().getHeaderName();
44)   }
45)
46)   public boolean  isOptional() {
47)     return getComplex().isOptional();
48)   }
49)
50)   public void setHeaderName(String headerName) {
51)     createComplex().setHeaderName(headerName);
52)   }
53)
54)   public void setOptional(boolean optional) {
55)     createComplex().setOptional(optional);
56)   }
57)
58)   public void setObject(Object object) {
59)     try{
60)       if(object instanceof Complex)
61)         super.setObject(object);
62)       else{
63)         throw new RuntimeException("The object " + object + " is not an Complex");
64)       }
65)     }
66)     catch (Throwable t){
67)       log.error(t);
68)     }
69)   }
70) }
```

If the `JavaAssertion` was used directly by the handler code we could also just use the `JavaAssertionMarshaller` directly to marshal the assertion between Java and XML. Since this example is using its own custom assertion class that wraps the `JavaAssertion` it will also need a custom marshaller.

```
01) public class ComplexAssertionMarshaller implements AssertionMarshaller {
02)
03)   private static QName[] supportedAssertions =
04)       new QName[] { ComplexPolicyConstants.COMPLEX_POLICY_NAME };
05)
06)   private JaxbAssertionMarshaller jaxbMarshaller;
07)
08)   public void setJaxbMarshaller(JaxbAssertionMarshaller jaxbMarshaller) {
09)         this.jaxbMarshaller = jaxbMarshaller;
10)     }
11)
12)   @Override
13)   public QName[] getSupportedAssertions() {
14)     return supportedAssertions;
15)   }
16)
17)   @Override
18)   public void marshal(Assertion assertion, Element element) throws GException {
19)     if(assertion instanceof ComplexAssertion) {
```

```
20)       ComplexAssertion complexAssertion = (ComplexAssertion)assertion;
21)       Complex complexPolicy = (Complex)complexAssertion.getObject();
22)       if (complexPolicy == null) { // in case it wasn't constructed completely
23)         complexPolicy = new Complex();
24)         complexAssertion.setObject(complexPolicy);
25)         complexAssertion.setName(ComplexPolicyConstants.COMPLEX_POLICY_NAME);
26)       }
27)       this.jaxbMarshaller.marshal(assertion, element);
28)     } else {
29)       throw new GException(PolicyErrorCode.UNSUPPORTED_ASSERTION);
30)     }
31)   }
32)
33)   @Override
34)   public Assertion unmarshal(Element element) throws GException {
35)     ComplexAssertion  complexAssertion = new ComplexAssertion();
36)     JavaAssertion javaAssertion =
37)         (JavaAssertion)this.jaxbMarshaller.unmarshal(element);
38)     if(javaAssertion.getObject() instanceof Complex) {
39)         Complex  complexPolicy = (Complex)javaAssertion.getObject();
40)       complexAssertion.setObject(complexPolicy);
41)     }
42)     else {
43)       throw new GException(PolicyErrorCode.UNSUPPORTED_ASSERTION);
44)     }
45)
46)     return  complexAssertion;
47)   }
48)
49)   @Override
50)   public Assertion unmarshal(Element element, Policy subPolicy)
51)       throws GException {
52)     throw new GException(PolicyErrorCode.SUB_POLICY_NOT_SUPPORTED);
53)   }
54) }
```

A `JaxbAssertionMarshaller` is embedded in this custom marshaller on line 6. The marshaller returns to the framework the assertions it supports on line 14. This tells the framework what XML elements to ask the marshaller to process.

On lines 18 – 31 the marshaller extracts the assertion information and creates the Complex JAXB object that it can then marshal to an XML element using the `JaxbAssertionMarshaller`.

On lines 34 – 47 the marshaller uses the `JaxbAssertionMarshaller` to marshal the XML element to a `Complex` JAXB object. It then wraps the output in a `ComplexAssertion` object.

The unmarshal method starting on line 50 is called when the framework detects the assertion has a nested policy. In this case on line 52 the marshaller throws an exception as this assertion does not support nested policies. It should not be called by the framework unless the policy was somehow created incorrectly.

There are two policy handlers in this example, one acting as a consumer of message exchanges and the other a provider. The source code for the provider is shown below.

```
01) public class ComplexPolicyProviderHandler implements MessageHandler {
02)
03)   // QName of missing header fault code
04)   private static final QName MISSING_HEADER_CODE =
05)       new QName(ComplexPolicyConstants.COMPLEX_POLICY_NS, "MissingHeader");
```

```
06)   // Message for missing header
07)   private static final String MISSING_HEADER_MSG = "Required header was missing";
08)   // QName of incorrect header content fault code
09)   private static final QName INVALID_HEADER_CODE =
10)       new QName(ComplexPolicyConstants.COMPLEX_POLICY_NS, "InvalidValue");
11)   // Message for incorrect header content
12)   private static final String INVALID_HEADER_MSG =
13)       "Header value does not match operation";
14)
15)   private String headerName;
16)   private boolean isOptional = true;
17)
18)   private static Log log = Log.getLog(ComplexPolicyProviderHandler.class);
19)
20)   public void setHeaderName(String headerName) {
21)      this.headerName = headerName;
22)   }
23)
24)   public void setOptional(boolean isOptional) {
25)      this.isOptional = isOptional;
26)   }
27)
28)   public void close(MessageContext context) {
29)     // no cleanup necessary
30)   }
31)
32)   /* Checks for the existence of the header and verifies the value matches the
33)    * current operation
34)    */
35)   public boolean handleMessage(MessageContext context)
36)       throws MessageFaultException {
37)     try {
38)      Header header = null;
39)      // get the current transport headers
40)      Headers headers = (Headers)context.getMessage().getProperty(
41)          MessageProperties.TRANSPORT_HEADERS);
42)      if (headers != null) {
43)        header = headers.getHeader(this.headerName);
44)        String operationName = context.getExchange().getOperationName();
45)        // if the header doesn't match the current operation flag as an error
46)        if (header != null && !header.getValue().equals(operationName)) {
47)          MessageFaultException mfe =
48)              new MessageFaultException(INVALID_HEADER_CODE, INVALID_HEADER_MSG);
49)          // set error so an alert is generated - must match alert code in PM
50)          mfe.setError(ComplexPolicyErrorCode.INVALID_HEADER_ERROR,
51)              new Object[] {operationName, this.headerName, header.getValue()});
52)          throw mfe;
53)        }
54)      }
55)      // if the header is mandatory but not present flag as an error
56)      if (!isOptional && header == null) {
57)        MessageFaultException mfe =
58)            new MessageFaultException(MISSING_HEADER_CODE, MISSING_HEADER_MSG);
59)        // set error so an alert is generated - must match alert code in PM database
60)        mfe.setError(ComplexPolicyErrorCode.MISSING_HEADER_ERROR,
61)            new Object[] {this.headerName});
62)        throw mfe;
63)      }
64)       return true; // continue handler processing
65)     } catch (Exception e) {
66)      log.error(e);
67)       throw new MessageFaultException(
68)           ComplexPolicyConstants.COMPLEX_FAULT_CODE, e.getLocalizedMessage());
```

```
69)     }
70)   }
71) }
```

The header name and optional flag from the policy assertion are private data members on lines 15 and 16. The handler does not read the assertion itself. That is the job of the factory seen later.

The `handleMessage()` method starting on line 35 is called by the framework to enforce the policy when receiving a request (IN) message. It is not called when processing a response (OUT) message because a handler is not created for the response message by the factory (see below).

On line 44 the handler retrieves the header with the name in the policy. On line 46 the handler compares the header value to the operation name. If they do not match a `MessageFaultException` is generated and thrown on lines 47 - 52. The exception tells the framework that policy enforcement has failed and to return a fault to the client with the code and message added to the exception.

On line 56 the handler checks to see if the header is not present and its presence is not optional. If this is the case it will generate a different `MessageFaultException` with a different code and message on lines 57 – 62.

If none of the checks fail, the handler indicates the message has passed policy enforcement by returning true on line 64.

The `close()` method on lines 28 - 30 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 for the consumer handler is shown next. Its purpose is to create a header with the name in the policy with the value of the current operation so the message passes enforcement at the downstream service.

```
01) public class ComplexPolicyConsumerHandler implements MessageHandler {
02)
03)     private static Log log = Log.getLog(ComplexPolicyConsumerHandler.class);
04)   private String headerName;
05)
06)   public void setHeaderName(String headerName) {
07)     this.headerName = headerName;
08)   }
09)
10)   public void close(MessageContext context) {
11)     // no cleanup necessary
12)   }
13)
14)   /* Inserts the operation name as an outbound transport header. */
15)   public boolean handleMessage(MessageContext context)
16)       throws MessageFaultException {
17)     try {
18)       // get the current outbound transport headers
19)       Headers headers = (Headers)context.getMessage().getProperty(
20)           MessageProperties.TRANSPORT_HEADERS);
21)       // may not be any yet, if that's the case create a new property for it
22)       if (headers == null) {
23)         headers = new BasicHeaders();
```

```
24)       }
25)       if (headers.containsHeader(this.headerName)) {
26)         /* if it's there it may be left over from the inbound side (see
27)          * preserve transport headers) and we must remove it
28)          */
29)         headers.removeHeader(this.headerName);
30)         // add the new header, get the operation name from the exchange
31)         headers.addHeader(
32)             this.headerName, context.getExchange().getOperationName());
33)       }
34)       return true; // continue handler processing
35)     } catch (Exception e) {
36)      log.error(e);
37)      throw new MessageFaultException(
38)              ComplexPolicyConstants.COMPLEX_FAULT_CODE, e.getLocalizedMessage());
39)     }
40)   }
41) }
```

The source code of the policy handler factory is below. Only one factory is needed to create both the provider and consumer policy handlers.

```
01) public class ComplexPolicyHandlerFactory extends SimplePolicyHandlerFactory {
02)
03)   // capability stating support for the policy
04)   private static PolicyHandlerFactoryCapability gCapability;
05)   static {
06)     gCapability = new PolicyHandlerFactoryCapability();
07)     gCapability.addSupportedAssertionNamespace(
08)         ComplexPolicyConstants.COMPLEX_POLICY_NS);
09)   }
10)
11)   protected MessageHandler create(Policy policy, HandlerContext context,
12)       HandlerRole role) throws GException {
13)     MessageHandler handler = null;
14)     Assertion complexAssert = getAssertion(policy);
15)     if (complexAssert != null) {
16)       // our marshaller returns a JavaAssertion holding a Complex object
17)       Complex complex = (Complex)((ComplexAssertion)complexAssert).getObject();
18)       // only if being called on provider side for an IN message we create a
19)       // provider handler
20)       if (role == HandlerRole.PROVIDER &&
21)           ((WSDLHandlerContext)context).getParameterType() == ParameterType.IN) {
22)         ComplexPolicyProviderHandler providerHandler =
23)             new ComplexPolicyProviderHandler();
24)         providerHandler.setHeaderName(complex.getHeaderName());
25)         providerHandler.setOptional(complex.isOptional());
26)         handler = providerHandler;
27)       // only if being called on consumer side for an IN message we create a
28)       // consumer handler
29)       } else if (role == HandlerRole.CONSUMER &&
30)           ((WSDLHandlerContext)context).getParameterType() == ParameterType.IN) {
31)         ComplexPolicyConsumerHandler consumerHandler =
32)             new ComplexPolicyConsumerHandler();
33)         consumerHandler.setHeaderName(complex.getHeaderName());
34)         handler = consumerHandler;
35)       }
36)     }
37)     return handler;
38)   }
39)
40)   /* Return the policy we support */
```

```
41)   public PolicyHandlerFactoryCapability getCapability() {
42)     return gCapability;
43)   }
44)
45)   /* Find the policy assertion we support, if present */
46)   private Assertion getAssertion(PolicyOperator po) {
47)     Assertion complexAssert = null;
48)
49)     // first check if present in policy operator's immediate child assertions
50)     for (Assertion assertion : po.getAssertions()) {
51)       if (assertion.getName().equals(ComplexPolicyConstants.COMPLEX_POLICY_NAME)) {
52)         complexAssert = assertion;
53)         break;
54)       }
55)     }
56)
57)     if (complexAssert == null) {
58)       for (PolicyOperator subPo : po.getPolicyOperators()) {
59)         if ((complexAssert = getAssertion(subPo)) != null) {
60)           break;
61)         }
62)       }
63)     }
64)     return complexAssert;
65)   }
66) }
```

The handler factory extends `SimplePolicyHandlerFactory` since there is no chance of getting top level policy choices.

On line 14 the assertion is extracted from the policy using the `getAssertion()` method on lines 46 - 65. That method recursively searches for an assertion with the Complex assertion's name.

On line 20 the check is made to see if a provider handler should be constructed and returned. If the role of the handler to be returned is `HandlerRole.PROVIDER` and the message the handler will process is the IN message then a provider handler should be returned.

One line 29 the check is made to see if a consumer handler should be constructed and returned. If the role of the handler to be returned is `HandlerRole.CONSUMER` and the message the handler will process is the IN message then a consumer handler should be returned. A common point of confusion is although the message being processed will be sent out of the container it is still the input message of the downstream service's operation so it is the IN message, not the OUT message.

BUNDLE

The classes described in the previous section need to be packaged in an OSGi Bundle so that they can be deployed to the SOA Container. The ComplexAssertionMarshaller and ComplexPolicyHandlerFactory need to be published as an OSGi service so that the Policy Handler Framework can load them. In this example, Blueprint is used to construct and publish the OSGi services using Spring. Spring and Blueprint are not requirements but is used here for simplicity.

The assertion marshaller is published using the following spring snippet.

```
01) <bean id="complex.assertion.marshaller"
class="com.soa.examples.policy.complex.assertion.marshaler.ComplexAssertionMarshaller"
>
02)   <property name="jaxbMarshaller" ref="complex.jaxb.marshaller"/>
03) </bean>
04)
05) <bean id="complex.jaxb.marshaller"
class="com.soa.policy.wspolicy.JaxbAssertionMarshaller"  init-method="init">
06)   <property name="assertionQNames">
07)     <list>
08)       <ref bean="complex.assertion.name"/>
09)     </list>
10)   </property>
11)   <property name="jaxbPaths">
12)     <list>
13)       <value>com.soa.examples.policy.complex.assertion.model</value>
14)     </list>
15)   </property>
16) </bean>
17)
18) <bean id="complex.assertion.name" class="javax.xml.namespace.QName">
19)   <constructor-arg
value="http://soa.com/products/policymanager/examples/policy/complex" />
20)   <constructor-arg value="Complex"/>
21) </bean>
22)
23) <osgi:service ref="complex.assertion.marshaller"
interface="com.soa.policy.wspolicy.AssertionMarshaller">
24)   <osgi:service-properties>
25)     <entry key="name" value="com.soa.examples.policy.complex.marshaller"/>
26)   </osgi:service-properties>
27) </osgi:service>
```

Lines 01 – 21 construct the `ComplexAssertionMarshaller` and all its dependencies. The `JaxbAssertionMarshaller` that is used within the `ComplexAssertionMarshaller` is constructed on lines 05 – 16.

The `ComplexAssertionMarshaller` is published as an OSGi service on lines 23 – 27. It must be published using the `AssertionMarshaller` interface. It is given a "name" property on line 25 as a good practice when publishing OSGi services. The name should be unique among all services published.

The `ComplexPolicyHandlerFactory` is published using the following Spring snippet. Because the policy handlers are validating and creating transport level headers the factory will be published with a "concrete" scope instead of "abstract." Although "abstract" is easier for defining policies that are independent of binding, not all bindings will have transport headers and there will definitely not be transport headers with a virtual service invokes another virtual service in the same container.

```
01) <bean id="complex.wsphandler.factory"
class="com.soa.examples.policy.complex.handler.ComplexPolicyHandlerFactory"/>
02)
03) <osgi:service  ref="complex.wsphandler.factory"
interface="com.soa.policy.wspolicy.handler.WSPHandlerFactory">
04)   <osgi:service-properties>
05)     <entry key="name" value="com.soa.examples.complex.in.http.wsp.factory"/>
06)     <entry key="scope" value="concrete"/>
07)     <entry key="binding" value="http"/>
08)     <entry key="role" value="provider"/>
```

```
09)   </osgi:service-properties>
10) </osgi:service>
11)
12) <osgi:service  ref="complex.wsphandler.factory"
interface="com.soa.policy.wspolicy.handler.WSPHandlerFactory">
13)   <osgi:service-properties>
14)     <entry key="name" value="com.soa.examples.complex.in.soap.wsp.factory"/>
15)     <entry key="scope" value="concrete"/>
16)     <entry key="binding" value="soap"/>
17)     <entry key="role" value="provider"/>
18)   </osgi:service-properties>
19) </osgi:service>
20)
21) <osgi:service  ref="complex.wsphandler.factory"
interface="com.soa.policy.wspolicy.handler.WSPHandlerFactory">
22)   <osgi:service-properties>
23)     <entry key="name" value="com.soa.examples.complex.out.http.wsp.factory"/>
24)     <entry key="scope" value="concrete"/>
25)     <entry key="binding" value="http"/>
26)     <entry key="role" value="consumer"/>
27)   </osgi:service-properties>
28) </osgi:service>
29)
30) <osgi:service  ref="complex.wsphandler.factory"
interface="com.soa.policy.wspolicy.handler.WSPHandlerFactory">
31)   <osgi:service-properties>
32)     <entry key="name" value="com.soa.examples.complex.out.soap.wsp.factory"/>
33)     <entry key="scope" value="concrete"/>
34)     <entry key="binding" value="soap"/>
35)     <entry key="role" value="consumer"/>
36)   </osgi:service-properties>
37) </osgi:service>
```

The creation of the `ComplexPolicyHandler` factory is simple and is done on line 01. Then that same factory instance is published to both the HTTP (REST) and SOAP bindings. Because the factory constructs handlers acting in both the "consumer" and "provider" roles it must be published multiple times with those roles. In all, the single factory instance is published as four OSGi services.

On lines 03 – 10 the factory is published as a provider side HTTP factory. On lines 12 – 19 the factory is published as a provider side SOAP factory. On lines 21 – 28 the factory is published as a consumer side HTTP handler. On lines 30 – 37 the factory is published as a consumer side SOAP handler.

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

```
01) Manifest-Version: 1.0
02) Bundle-ManifestVersion: 2
03) Bundle-Name: SOA Software Complex Policy Handler Example
04) Bundle-SymbolicName: com.soa.examples.policy.handler.complex
05) Bundle-Version: 7.0.0
06) Bundle-Vendor: SOA Software
07) Import-Package: com.digev.fw.exception;version="7.0.0",
08)  com.digev.fw.log;version="7.0.0",
09)  com.soa.message;version="7.0.0",
10)  com.soa.message.handler;version="7.0.0",
11)  com.soa.message.handler.wsdl;version="7.0.0",
```

```
12)  com.soa.message.header;version="7.0.0",
13)  com.soa.message.header.impl;version="7.0.0",
14)  com.soa.policy;version="7.0.0",
15)  com.soa.policy.template;version="7.0.0",
16)  com.soa.policy.wspolicy;version="7.0.0",
17)  com.soa.policy.wspolicy.handler;version="7.0.0",
18)  com.soa.policy.wspolicy.handler.ext;version="7.2.0",
19)  javax.xml.bind,
20)  javax.xml.bind.annotation,
21)  javax.xml.namespace,
22)  org.w3c.dom
23) Export-Package: com.soa.examples.policy.complex,
24)  com.soa.examples.policy.complex.assertion,
25)  com.soa.examples.policy.complex.assertion.model,
26)  com.soa.examples.policy.complex.template
```

Lines 01 – 06 hold general information about the Bundle. Lines 07 – 22 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.

Lines 23 – 26 list the packages that are exported, or published, to other bundles loaded in the system. This is required for the Policy Handler Framework to be able to load the assertion classes as they are constructed using a JAXB context from another bundle and could possibly be used by a user interface bundle for displaying the policy in the Policy Manager Management Console.

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 Management 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 and assertion marshallers will be picked up by the Policy Handler Framework.

References

[WS-Policy]
D. Box, et al, *"Web Services Policy Framework (WS-Policy),"* April 2006. (See http://www.w3.org/Submission/2006/SUBM-WS-Policy-20060425/)

[WS-PolicyAttachment]
D. Box, et al, *"Web Services Policy Attachment (WS-PolicyAttachment),"* April 2006. (See http://www.w3.org/Submission/2006/SUBM-WS-PolicyAttachment-20060425/)