

nehta

Example Technical Implementation of Interoperable Web Services

JAX-WS

Version 1.0 — 3 July 2007

Public release

National E-Health Transition Authority Ltd

Level 25

56 Pitt Street

Sydney, NSW, 2000

Australia.

www.nehta.gov.au**Disclaimer**

NEHTA makes the information and other material (“Information”) in this document available in good faith but without any representation or warranty as to its accuracy or completeness. NEHTA cannot accept any responsibility for the consequences of any use of the Information. As the Information is of a general nature only, it is up to any person using or relying on the Information to ensure that it is accurate, complete and suitable for the circumstances of its use.

Document Control

This document is maintained in electronic form. The current revision of this document is located on the NEHTA Web site and is uncontrolled in printed form. It is the responsibility of the user to verify that this copy is of the latest revision.

Copyright © 2007, NEHTA.

This document contains information which is protected by copyright. All Rights Reserved. No part of this work may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without the permission of NEHTA. All copies of this document must include the copyright and other information contained on this page.

Table of contents

1	Introduction	1
1.1	Background.....	1
1.2	Purpose.....	1
1.3	Scope	1
1.4	Intended audience	1
1.5	Definitions, acronyms, abbreviations.....	1
1.5.1	Terminology	2
1.6	Style Conventions	2
1.7	Overview	3
2	Example service.....	4
2.1	Organisational	4
2.1.1	Testing	4
2.1.2	Sending discharge summaries.....	5
2.1.3	Checking discharge summary status	5
2.2	Informational	5
2.2.1	Discharge summary.....	5
2.2.2	Status	6
2.3	Technical	6
2.3.1	Informational attributes	6
2.3.2	Behavioural attributes.....	6
2.3.3	Non-functional attributes.....	11
3	Overview of JAX-WS	12
3.1	General background	12
3.2	Technical overview.....	12
3.2.1	Client	12
3.2.2	Server	12
3.3	Recommended version.....	13
3.4	Requirements.....	13
3.4.1	Client deployment	13
3.4.2	Web service deployment	13
3.4.3	Client development.....	13
3.4.4	Web service development.....	13
3.5	Platform used.....	14
4	Web service client	15
4.1	Specify the policies in the WSDL files	15
4.1.1	WS-Addressing	16
4.1.2	WS-Security	17
4.1.3	Specifying keys and certificates	22
4.2	Generate the service interface classes from the WSDL files	23
4.3	Implement the client application	24
4.3.1	Create a service object.....	25
4.3.2	Get a port object.....	25
4.3.3	Set request context properties on the port	25
4.3.4	Invoke Web service operations.....	27
4.4	Implement a SOAP handler	27
4.5	Implement the security callback handler	28
5	Web service	32
5.1	Specify the policies in the WSDL	32
5.2	Generate the service interface classes from the WSDL files	32
5.3	Implement the Web service.....	33
5.4	Implement the security callback handler	33

5.5	Package the Web service	34
5.5.1	Directory structure	34
5.5.2	Create the configuration files	34
5.5.3	Create the WAR file	35
5.6	Deploy the Web service	36
5.6.1	Install JAX-WS on the server	36
5.6.2	Configure the server for the Web service	36
5.6.3	Deploy the WAR file.....	36
5.6.4	Accessing the Web service.....	36
Appendix A: References.....		37
Appendix B: Installation.....		38
B.1	Java Development Kit.....	38
B.2	Java Cryptography Extension Unlimited Strength Jurisdiction Policy Files	38
B.3	Ant.....	38
B.4	JAX-WS.....	38
B.5	Servlet container	39
B.6	Sun Java System Application Server (SJSAS).....	39
B.6.1	Specify System properties.....	40
Appendix C: Key management.....		41
C.1	Key store types	41
C.2	Tools	41
Appendix D: Debugging.....		42
D.1	Use a SOAP handler	42
D.2	Use an HTTP tool	43
D.3	View server logs.....	44

Document information

Change history

Version	Date	Comments
1.0	2007-07-03	Public release

This page is intentionally left blank.

1 Introduction

1.1 Background

The National E-Health Transition Authority (NEHTA) has recommended Web services as the technology for application-to-application communications for Australia's e-health national infrastructure.

NEHTA has published a number of technical documents to support the use of Web services. These include the *Web Services Standards Profile* [WSSP2006] and the *Guidelines for Implementing Interoperable Web Services* [GIIWS2007].

1.2 Purpose

This document provides an example of how Web services conforming to the Guidelines can be implemented using a specific toolkit: Java API for XML Web Services (JAX-WS) [JAXWS].

Its main purpose is to support the understanding and interpretation of the guidelines. However, it can also assist programmers who are learning how to use JAX-WS.

This document is provided for educational purposes only. The method it describes is only one approach; there might be other, equally valid approaches. The code samples in this document are designed for simplicity and ease of understanding, rather than robustness and reuse. They are not written for use in a production system.

1.3 Scope

This document only covers JAX-WS. Also available are example technical implementation documents covering .NET Web Services Enhancements (WSE) 3.0 [WSE] and .NET Windows Communication Foundation (WCF) [WCF]. Those other example technical implementations can interoperate with this implementation, but they will not be discussed in this document.

These examples are not an endorsement of these platforms by NEHTA.

1.4 Intended audience

This document is intended for:

- Software developers; and
- System administrators.

It is expected that the reader is familiar with programming using Java, and has an understanding of Web services and Public Key Infrastructure (PKI) security using X.509 certificates.

The reader is also expected to be familiar with the NEHTA *Web Services Standards Profile* [WSSP2006] and the *Guidelines for Implementing Interoperable Web Services* [GIIWS2007]. The guidelines from [GIIWS2007] are referred to by their guideline number (e.g. "Guideline TG2").

1.5 Definitions, acronyms, abbreviations

API	Application programming interface
CA	Certificate authority
DOM	Document Object Model

HTTP	Hypertext Transfer Protocol
J2SE	Java 2 Standard Edition
J2EE	Java 2 Enterprise Edition
JAR	Java Archive
JAXB	Java API for XML Binding
JAX-RPC	Java API for XML-based RPC
JAX-WS	Java API for XML Web Services
JDK	Java Development Kit
JRE	J2SE Runtime Environment
RPC	Remote Procedure Call
SJSAS	Sun Java System Application Server
WAR	Web Archive
WCF	Windows Communication Foundation
WSDL	Web Service Definition Language
WSE	Web Services Enhancements
WSIT	Web Services Interoperability Technologies

1.5.1 Terminology

This document uses the following terms:

Web services	The technology for communicating between computer applications using SOAP, WSDL and other related standards.
Web service	A computer program that provides services and uses the Web services technologies to allow access to those services.
Web service client	A computer program that uses the services provided by a Web service. It invokes operations that are provided by the Web service. The abbreviated term "client" can also be used.
Web server	A computer program that makes Web resources (predominantly HTML Web pages) available via Web protocols (predominantly HTTP).
Server	A computer that is hosting a Web server or other programs that provides a service to other programs.

1.6 Style Conventions

This document uses the following style conventions:

<i>Italics</i>	<ul style="list-style-type: none"> • Document titles • Program names, tool names • File names, directory paths • URLs • Keywords
Monospace	<ul style="list-style-type: none"> • XML fragments, namespaces, names of XML elements and types • Code fragments, names of classes, methods and

	fields <ul style="list-style-type: none">• Assemblies, packages• Command-line calls and arguments• Configuration properties
Monospace + Bold	<ul style="list-style-type: none">• Emphasis within XML and code fragments
"Double quotes"	<ul style="list-style-type: none">• Graphical user interface options

1.7 Overview

Chapter 2 describes the service used as an example for this document.

Chapter 3 provides a brief overview of JAX-WS.

Chapter 4 describes how to create a Web service client using JAX-WS.

Chapter 5 describes how to create a Web service using JAX-WS.

Appendix A lists references.

Appendix B provides instructions and notes on software installation.

Appendix C provides information on security key management.

Appendix D provides tips on debugging JAX-WS programs.

2 Example service

This chapter describes the example service that will be implemented.¹

The specification of a service would normally be produced by an independent organisation, which brings together the requirements of all the stakeholders. This chapter is an abridged version of the service specification that would be produced—since this document is concerned with programming Web services, it focuses on the WSDL specification.

The example technical implementation described in this document assumes a WSDL-first approach, where the Web service implementation is developed using classes generated from the WSDL. This approach is in contrast to the implementation-first approach, where the WSDL is automatically generated from the implementation code. The WSDL-first approach is more applicable to an interoperable e-health environment, where standard WSDL specifications developed by independent organisations should be used to build Web services.

The structure of this chapter follows the approach described by the NEHTA *Interoperability Framework* [NIF2006] and uses concepts from the *Technical Architecture for Implementing Services* [TAIS2006].

2.1 Organisational

This example scenario is based on the exchange of discharge summaries. It has been simplified for ease of understanding—it is not intended to be a real world discharge summary scenario.

In the community for discharge summary exchange, there are two roles:

- Sending provider: the program that generates the discharge summary and sends it; and
- Receiving provider: the program that receives the discharge summary and tracks whether it has been acknowledged.

The business process of sending a discharge summary involves three activities:

- Testing if the receiving provider's discharge summary receiving service is operating;
- Sending discharge summaries from a sending provider to a receiving provider; and
- Checking the status of a discharge summary to see if the receiving provider has processed and acknowledged it.

2.1.1 Testing

In this activity, one party wishes to determine whether the receiving provider's service is operational or not. It can be used to check if the programs and the network have been correctly configured.

This activity illustrates the use of an operation that requires no parameters. It is implemented as an operation that does nothing, other than to return an empty result.

This operation is called "ping" after the program used to test if an internet protocol host is reachable across an IP network [PING].

¹ This chapter is identical to the corresponding chapter in the other *Example Technical Implementation of Interoperable Web Services* documents (i.e. for WCF and WSE 3.0.)

This is a request-response operation at the technical-level to comply with Guideline TG4 from NEHTA's *Guidelines for Implementing Interoperable Web Services* [GIIWS2007].

2.1.2 Sending discharge summaries

In this activity, a sending provider creates a discharge summary and sends it to the receiving provider.

When the discharge summary has been received, the receiving provider keeps track of which discharge summary it has received and whether it has been acknowledged by a person at the receiving organisation. This behaviour is to support the checking operation described in section 2.1.3.

This is a one-way operation at the business-level, and no response data is returned to the sender. The only way the sender can discover if it was successfully received is to use the check discharge summary status operation.

At the technical-level, this operation is implemented as a request-response operation to comply with Guideline TG4. That is, a response is sent back, but it contains no business-level information.

2.1.3 Checking discharge summary status

In this activity, a sending provider queries the receiving provider about the status of a particular discharge summary. The receiving provider returns the result to the querying provider.

This is a request-response operation: a response containing the status is returned.

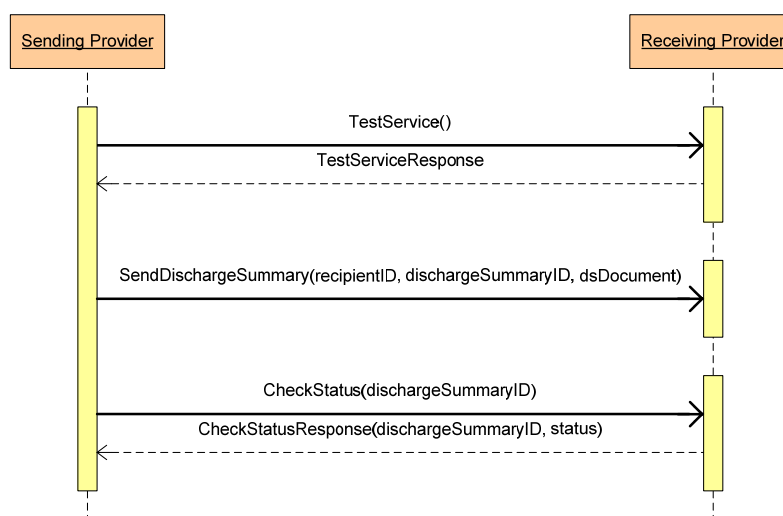


Figure 1: Example discharge summary business workflows

2.2 Informational

2.2.1 Discharge summary

The discharge summary is modelled as a document with an identifier and a notes field. The document identifier should be a globally unique string that is allocated by the sender of the discharge summary. The notes field contains unstructured text.

The discharge summary data model is simple since the aim of this example is to demonstrate Web services, rather than demonstrate a real discharge summary scenario. NEHTA's *National Discharge Summary Data Content Specification* [NDS2006] contains much more structured data and metadata in the data model of a discharge summary.

2.2.2 Status

The possible status values for a discharge summary are:

- Not received: a discharge summary with the given document identifier has not been received;
- Pending acknowledgement: it has been received, but has not been acknowledged by the receiving party; and
- Acknowledged: it has been received and acknowledged.

The delay between receiving a discharge summary and it being acknowledged is not defined by the service. This is because acknowledgement is a manual process involving a person—it could take minutes or days to perform.

2.3 Technical

This section describes the technical aspects of the service interface specification. It is organised using the three types of attributes, as defined in the Technical Architecture: informational, behavioural and non-functional attributes [TAIS2006].

2.3.1 Informational attributes

The XML Schema used to define a discharge summary document is shown below. It defines a single complex type with 2 child elements: `DocumentId` and `Notes`.

This XSD file will be stored in a file called *DischargeSummary.xsd*.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://example.org/ns/2007/DischargeSummary/v1"
  elementFormDefault="qualified">

  <xsd:complexType name="DischargeSummaryType">
    <xsd:sequence>
      <xsd:element name="DocumentId" type="xsd:string"/>
      <xsd:element name="Notes" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

2.3.2 Behavioural attributes

The WSDL file contains a formal specification of the behavioural aspects of the service interface.

A WSDL file is not the complete documentation for a real service, which would require additional documentation to fully describe the service's behavioural attributes.

However, only the WSDL will be provided for this simple test service, because a complete description of the service is not required for the level of interoperability testing that it will be used for.

2.3.2.1 WSDL containing service interface information

Guideline TG1 recommends separating the service interface information from the service instance information. This section will go through the WSDL containing the service interface information for our sample Web service.

This WSDL file will be stored in a file called *DischargeSummaryReceiverInterface.wsdl*.

2.3.2.1.1 Header

The beginning of the WSDL file contains the start tag of the root element, which contains all the XML namespaces that this document will use.

The service namespace for this service was arbitrarily chosen to be:

```
http://example.org/ns/2007/DischargeSummaryReceiver/v1
```

Following Guideline TG7, this namespace will be used for the `targetNamespace` of the WSDL. It is associated with the namespace prefix of `tns` so that it can be referenced in the document. The prefix of the target namespace does not necessarily have to be `tns`; it is just a commonly used convention.

```
<?xml version="1.0"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
  xmlns:tns="http://example.org/ns/2007/DischargeSummaryReceiver/v1"
  targetNamespace="http://example.org/ns/2007/DischargeSummaryReceiver/v1"
  name="DischargeSummaryReceiver">
```

2.3.2.1.2 Types

The types section of the WSDL declares the elements and data types of the messages used by the service.

The definition of the discharge summary is imported from an external XML Schema file. This file was described in section 2.3.1.

```
<wsdl:types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://example.org/ns/2007/DischargeSummaryReceiver/v1"
    xmlns:ds="http://example.org/ns/2007/DischargeSummary/v1"
    targetNamespace="http://example.org/ns/2007/DischargeSummaryReceiver/v1"
    elementFormDefault="qualified">

    <xsd:import namespace="http://example.org/ns/2007/DischargeSummary/v1"
      schemaLocation="DischargeSummary.xsd"/>
```

The request and response elements for the ping operation are defined below. Since this operation takes no parameters and returns no results, both of these elements have an empty content model and no attributes.

To conform to Guideline TG3, this WSDL follows the wrapped convention. Thus, for all operations in this WSDL, the request element's name matches the operation's name, and the response element's name is the operation's name with a "Response" suffix.

```
<xsd:element name="Ping">
  <xsd:complexType/>
</xsd:element>

<xsd:element name="PingResponse">
  <xsd:complexType/>
</xsd:element>
```

The request and response elements for the send discharge summary operation are defined below.

The request is an element that contains the discharge summary document.

Although the send discharge summary operation requires no business-level response, it has a response element, which has an empty content model and no attributes. This operation is modelled as a request-response operation at the technical level to satisfy Guideline TG4.

```
<xsd:element name="SendDischargeSummary">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Document" type="ds:DischargeSummaryType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

```
<xsd:element name="SendDischargeSummaryResponse">
  <xsd:complexType/>
</xsd:element>
```

The request and response elements for the check status operation are defined below.

```
<xsd:element name="CheckStatus">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="DocumentId" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="CheckStatusResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Response" type="tns:ReceivedStatusType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

The following simple type defines the enumerated set of status values that could be returned by the check status operation.

```
<xsd:simpleType name="ReceivedStatusType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="NotReceived"/>
    <xsd:enumeration value="PendingAcknowledgement"/>
    <xsd:enumeration value="Acknowledged"/>
  </xsd:restriction>
</xsd:simpleType>
```

The send discharge summary and check status operations can return a fault. The structure of this fault element is defined below.

```
<xsd:element name="InvalidIdFault">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="FaultDescription" type="xsd:string"/>
      <xsd:element name="DocumentId" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
</wsdl:types>
```

2.3.2.1.3 Messages

The messages section of the WSDL identifies the messages used by the three operations of the service.

The messages follow the wrapped convention to conform to Guideline TG3. The messages only have one part, where each part references an XML Schema element that was declared in the types section of the WSDL.

```
<wsdl:message name="PingInMsg">
  <wsdl:part name="body" element="tns:Ping"/>
</wsdl:message>

<wsdl:message name="PingOutMsg">
  <wsdl:part name="body" element="tns:PingResponse"/>
</wsdl:message>

<wsdl:message name="SendDischargeSummaryInMsg">
  <wsdl:part name="body" element="tns:SendDischargeSummary"/>
</wsdl:message>

<wsdl:message name="SendDischargeSummaryOutMsg">
  <wsdl:part name="body" element="tns:SendDischargeSummaryResponse"/>
</wsdl:message>

<wsdl:message name="CheckStatusInMsg">
  <wsdl:part name="body" element="tns:CheckStatus"/>
</wsdl:message>
```

```

<wsdl:message name="CheckStatusOutMsg">
  <wsdl:part name="body" element="tns:CheckStatusResponse" />
</wsdl:message>

<wsdl:message name="InvalidIdFault">
  <wsdl:part name="fault" element="tns:InvalidIdFault" />
</wsdl:message>

```

2.3.2.1.4 Port Type

The portType section of the WSDL specifies the three operations in the service.

In each operation, all the input messages, output messages and faults are assigned a WS-Addressing Action. This follows Guideline TG5. The values used for the WS-Addressing Action follow the scheme set out in Guideline TG7.

```

<wsdl:portType name="DischargeSummaryReceiver">

  <wsdl:operation name="Ping">
    <wsdl:input message="tns:PingInMsg"
wsaw:Action="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummaryReceiver/PingRequest" />
    <wsdl:output message="tns:PingOutMsg"
wsaw:Action="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummaryReceiver/PingResponse" />
  </wsdl:operation>

  <wsdl:operation name="SendDischargeSummary">
    <wsdl:input message="tns:SendDischargeSummaryInMsg"
wsaw:Action="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummaryReceiver/SendDischargeSummaryRequest" />
    <wsdl:output message="tns:SendDischargeSummaryOutMsg"
wsaw:Action="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummaryReceiver/SendDischargeSummaryResponse" />
    <wsdl:fault name="InvalidIdFault" message="tns:InvalidIdFault"
wsaw:Action="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummaryReceiver/SendDischargeSummary/Fault/InvalidIdFault" />
  </wsdl:operation>

  <wsdl:operation name="CheckStatus">
    <wsdl:input message="tns:CheckStatusInMsg"
wsaw:Action="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummaryReceiver/CheckStatusRequest" />
    <wsdl:output message="tns:CheckStatusOutMsg"
wsaw:Action="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummaryReceiver/CheckStatusResponse" />
    <wsdl:fault name="InvalidIdFault" message="tns:InvalidIdFault"
wsaw:Action="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummaryReceiver/CheckStatus/Fault/InvalidIdFault" />
  </wsdl:operation>

</wsdl:portType>
</wsdl:definitions>

```

2.3.2.2 WSDL containing service instance information

The second WSDL file contains the service instance information. It defines how the abstract port type, which was defined in *DischargeSummaryReceiverInterface.wsdl*, maps to a particular technology—namely SOAP 1.2.

This second WSDL will be stored in a file called *DischargeSummaryReceiver.wsdl*.

2.3.2.2.1 Header

In the second WSDL file, the start tag of the root element again contains all the XML namespaces that this document will use. The *Web Services Standards Profile* [WSSP2006] recommends the use of SOAP 1.2 as the messaging protocol. This is specified in the WSDL by using the XML namespace for the SOAP 1.2 binding, which is namely:

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

This second WSDL should have a target namespace that matches the service namespace, namely:

```
http://example.org/ns/2007/DischargeSummaryReceiver/v1
```

The `tns` prefix is again used to refer to the target namespace.

Since it refers to the port type defined in the WSDL containing the service interface information, it must import the first WSDL file.

```
<?xml version="1.0"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:tns="http://example.org/ns/2007/DischargeSummaryReceiver/v1"
  targetNamespace="http://example.org/ns/2007/DischargeSummaryReceiver/v1"
  name="DischargeSummaryReceiver">

  <wsdl:import namespace="http://example.org/ns/2007/DischargeSummaryReceiver/v1"
    location="DischargeSummaryReceiverInterface.wsdl"/>
```

2.3.2.2.2 Binding

The binding section indicates the message format and protocol details for the abstract port types.

HTTP 1.1 is the transport protocol that is recommended by the *Web Services Standards Profile* [WSSP2006]. This is specified in the WSDL by setting the transport attribute of the SOAP 1.2 binding element to:

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

To comply with Guideline TG6, `soapAction` values are assigned to every operation. The value used for this `soapAction` follows the convention described in Guideline TG7.

To comply with Guideline TG2, the *document/literal* style is used. This is done by setting the style attributes of SOAP operation elements to `document` and the use attributes of SOAP body and fault elements to `literal`.

```
<wsdl:binding name="DischargeSummaryReceiverBinding"
  type="tns:DischargeSummaryReceiver">

  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>

  <wsdl:operation name="Ping">
    <soap:operation style="document"

soapAction="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummar
yReceiver/PingRequest"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>

  <wsdl:operation name="SendDischargeSummary">
    <soap:operation style="document"

soapAction="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummar
yReceiver/SendDischargeSummaryRequest"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
    <wsdl:fault name="InvalidIdFault">
      <soap:fault name="InvalidIdFault" use="literal"/>
    </wsdl:fault>
  </wsdl:operation>

  <wsdl:operation name="CheckStatus">
```

```

        <soap:operation style="document"
soapAction="http://example.org/ns/2007/DischargeSummaryReceiver/v1/DischargeSummaryReceiver/CheckStatusRequest"/>
        <wsdl:input>
            <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output>
            <soap:body use="literal"/>
        </wsdl:output>
        <wsdl:fault name="InvalidIdFault">
            <soap:fault name="InvalidIdFault" use="literal"/>
        </wsdl:fault>
    </wsdl:operation>
</wsdl:binding>

```

2.3.2.2.3 Service

The service part of the WSDL defines a service with concrete ports that are associated with a particular binding.

An address must be provided for the Web service instance. However, it is not necessary to provide an actual hard-coded URL. This address value can be overridden by the toolkit.

```

<wsdl:service name="DischargeSummaryReceiverService">
    <wsdl:port name="DischargeSummaryReceiver"
        binding="tns:DischargeSummaryReceiverBinding">
        <soap:address location="http://DUMMY_VALUE"/>
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

2.3.3 Non-functional attributes

This example service will follow all the Guidelines relating to non-functional attributes. These guidelines are:

- Guideline TG8: Include a WS-Security timestamp;
- Guideline TG9: Sign SOAP messages;
- Guideline TG10: Encrypt SOAP messages;
- Guideline TG11: Sign before encrypting the SOAP message;
- Guideline TG12: Use WS-SecurityPolicy's Basic256 algorithm suite;
- Guideline TG13: Use of Direct Reference and Subject Key Identifier for X.509 certificates;
- Guideline TG14: Provide WS-Addressing To, From, Action and MessageID headers in SOA request messages; and
- Guideline TG15: Provide WS Addressing Action, MessageID and RelatesTo headers in SOAP response messages.

Conformance to these guidelines is achieved through the WSDL. It is done through how the Web service client and Web service is implemented, which is the subject of chapters 4 and 5, rather than as part of the WSDL definition.

3 Overview of JAX-WS

3.1 General background

JAX-WS is a standard Java API for Web services, which was developed by the Java Community Process [JSR224].

There can be multiple implementations of JAX-WS API. This document will be based around JAX-WS RI, an open-source reference implementation of the JAX-WS specification. In general, the concepts described in this document will be relevant to other implementations of the JAX-WS API.

In this document, the term “JAX-WS” will be used to refer to the JAX-WS reference implementation and not the API specification.

3.2 Technical overview

3.2.1 Client

Using JAX-WS, a Web service client does not have to create and manipulate XML SOAP messages directly. The client can invoke an operation on a remote Web service by making a Java method call.

JAX-WS comes with a tool called *wsimport*, which generates Java classes from a given WSDL. It generates:

- Java class for the service declared in the WSDL;
- Java interfaces containing methods for the Web service operations of the ports defined in the WSDL;
- Java beans for the XML Schema types defined in the WSDL; and
- Exceptions for the SOAP faults declared in the WSDL.

This document will refer to these Java interfaces and classes generated by *wsimport* as “service interface classes”.

When the client makes a method call using these generated service interface classes, JAX-WS uses the Java API for XML Binding (JAXB) to marshall the Java method call and its parameters as a SOAP request. (JAXB is a standard Java technology that maps Java objects to and from XML.) JAX-WS might also manipulate the SOAP request before it is sent, based on configuration details. For instance, it can add addressing and security headers to the request, and sign and encrypt the request. When a SOAP response is returned to the client, JAX-WS verifies and decrypts the response if security is configured. JAXB then unmarshalls the SOAP response into a Java object that is returned to the client.

3.2.2 Server

On the server-side, the *wsimport* tool is also used to generate service interface classes. The Web service developer then codes a Java class that implements the generated Java interface containing the Web service methods. These classes and other supporting files are packaged into a Web archive (WAR) and deployed in a Web container.

When a SOAP request comes in, a servlet provided by the JAX-WS implementation processes the request. If security is configured, JAX-WS will verify and decrypt the SOAP request. JAXB unmarshalls the SOAP request. The appropriate method is then called on the Web service implementation class. The method's return value is marshalled by JAXB as a SOAP response, which might be signed and encrypted, before it is passed to the servlet to return over the network.

3.3 Recommended version

There are a number of JAX-WS versions. From an interoperability perspective, it is recommended that a JAX-WS version containing Web Services Interoperability Technologies (WSIT) be used. WSIT technology is designed for interoperability with Microsoft's .NET framework, in particular the WCF Web services stack in .NET 3.0. WSIT is not available as a separate plugin; it is bundled with JAX-WS in a release.

This document describes how to build and configure Web services and clients for the *JAX-WS 2.1 with WSIT* version. The code and configuration samples in this document might not work for other JAX-WS versions. For instance, *JAX-WS 2.1.1* does not recognise the security configuration described in this document.

In particular, *Milestone Release 4* of the *JAX-WS 2.1 with WSIT* version was used to build this example implementation and test it with other example implementations in .NET WCF and WSE 3.0.

3.4 Requirements

This section lists the software requirements. See *Appendix B: Installation* for installation instructions and notes.

3.4.1 Client deployment

- JAX-WS 2.1 with WSIT
- Java Runtime Environment (JRE) 5 or later
- Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files

3.4.2 Web service deployment

The same software is required as the client deployment as well as a servlet container to host the Web service.

Note: This document describes hosting a Web service in terms of the Sun Java System Application Server (SJSAS) because it is part of the platform used in testing the code and configuration fragments in the document. However, SJSAS is not required to host JAX-WS Web services. JAX-WS can be used with lightweight Java Web containers, such as Apache Tomcat, as well as other Java 2 Enterprise Edition (J2EE) application servers.

3.4.3 Client development

- JAX-WS 2.1 with WSIT
- Java Standard Edition Development Kit (JDK) 5 or later

3.4.4 Web service development

The same software is required as the client development.

3.5 Platform used

The code and configuration fragments in this document were tested using the following software:

- JAX-WS 2.1 with WSIT - Milestone Release 4
- SJSAS Platform Edition 9 Update 1
- JDK and JRE 5
- JCE Unlimited Strength Jurisdiction Policy Files 5
- Ant 1.6.5
- Windows XP

4 Web service client

This chapter describes how to build a Web service client using JAX-WS. The aim of a Web service client is to invoke an operation on a service instance.

Normally, the process of creating a Web service client simply involves generating the service interface classes from the WSDL files and then using those classes in the client program. However, to implement some of the interoperability features, a SOAP handler and a custom security callback handler are required.

The steps for building a Web service client are:

1. Specify the policies in the WSDL files;
2. Generate the service interface classes from the WSDL files;
3. Implement the client application;
4. Implement a SOAP handler; and
5. Implement the security callback handler.

4.1 Specify the policies in the WSDL files

In the WSIT version of JAX-WS, addressing and security for Web services are configured using WS-Policy in the WSDL.

Section 2.3.1 describes the WSDL files for the example service. In keeping with Guideline TG1 from NEHTA's *Guidelines for Implementing Interoperable Web Services* [GIIWS2007], the WSDL definition was separated into two files: the service interface WSDL and the service instance WSDL. The Web service developer will have to edit the service instance WSDL (described in section 2.3.2.1) to add the policy information to configure JAX-WS. The service interface WSDL (described in section 2.3.2.2) can be imported without any modifications to the updated service instance WSDL.

Policies are defined using WS-Policy's `Policy` element. Every policy should be given a unique identifier, as specified in the `wsu:Id` attribute of the `Policy` element. Policies can then be applied to a WSDL subject (e.g. a binding, a message, or a service) using WS-Policy's `PolicyReference` element.

In this case, all the policies will be embedded in the WSDL file. The policy references will then refer to the named policies using a relative URL consisting of just a fragment identifier (i.e. a hash followed by the policy name).

The general structure of the service WSDL file with the policy information is shown below. Notice how `PolicyReference` elements have been added to a number of WSDL subjects, and that the `Policy` elements are also included in the file.

```
<?xml version="1.0"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
  utility-1.0.xsd"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
  xmlns:tns="http://example.org/ns/2007/DischargeSummaryReceiver/v1"
  targetNamespace="http://example.org/ns/2007/DischargeSummaryReceiver/v1"
  name="DischargeSummaryReceiver">

  <wsdl:import namespace="http://example.org/ns/2007/DischargeSummaryReceiver/v1"
    location="DischargeSummaryReceiverInterface.wsdl"/>

  <wsdl:binding name="DischargeSummaryReceiverBinding"
    type="tns:DischargeSummaryReceiver">
    <!-- Apply 'AddressingPolicy', 'SecurityPolicy' and 'SecurityConfigPolicy'
```

```

        policies to the WSDL binding -->
<wsp:PolicyReference URI="#AddressingPolicy"/>
<wsp:PolicyReference URI="#SecurityPolicy"/>
<wsp:PolicyReference URI="#SecurityConfigPolicy"/>

<soap:binding .../>

...
<wsdl:operation name="SendDischargeSummary">
  <soap:operation .../>
  <wsdl:input>
    <soap:body .../>
    <!-- Apply 'SecureMessagePolicy' policy to operation's input message -->
    <wsp:PolicyReference URI="#SecureMessagePolicy"/>
  </wsdl:input>
  <wsdl:output>
    <soap:body .../>
    <!-- Apply 'SecureMessagePolicy' policy to operation's output message -->
    <wsp:PolicyReference URI="#SecureMessagePolicy"/>
  </wsdl:output>
  <wsdl:fault name="InvalidIdFault">
    <soap:fault .../>
    <!-- Apply 'SecureMessagePolicy' policy to operation's fault message -->
    <wsp:PolicyReference URI="#SecureMessagePolicy"/>
  </wsdl:fault>
</wsdl:operation>
...
</wsdl:binding>

<wsdl:service ...>
  ...
</wsdl:service>

<!-- Define the policies -->
<wsp:Policy wsu:Id="AddressingPolicy"> ...
</wsp:Policy>

<wsp:Policy wsu:Id="SecurityPolicy"> ...
</wsp:Policy>

<wsp:Policy wsu:Id="SecureMessagePolicy"> ...
</wsp:Policy>

<wsp:Policy wsu:Id="SecurityConfigPolicy"> ...
</wsp:Policy>
</wsdl:definition>

```

Four different policies are defined and used. These will be described in the following subsections.

Note: The policy information in the WSDL files is used at run-time to configure Web service calls. It does not affect the generation of service interface classes. Therefore, this step does not necessarily have to be done first—it just needs to be done before the Web service client is run.

4.1.1 WS-Addressing

JAX-WS can be configured to include WS-Addressing 1.0 information in SOAP messages by including the `UsingAddressing` element in a policy and referencing the policy in the WSDL binding.

A policy called `AddressingPolicy` is defined, and is used in the binding for the service. The name `AddressingPolicy` is arbitrary—any unique name could have been used.

The complete policy is shown below:

```

<wsdl:definitions ...>
  ...

  <wsdl:binding name="DischargeSummaryReceiverBinding" type="...">
    <wsp:PolicyReference URI="#AddressingPolicy"/>
    ...
  </wsdl:binding>
  ...
</wsdl:definitions>

```

```

<wsp:Policy wsu:Id="AddressingPolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsaw:UsingAddressing/>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
...
</wsdl:definition>

```

This policy will cause JAX-WS to add WS-Addressing information that conforms to Guideline TG14 with a few exceptions. These exceptions will be resolved using a SOAP handler, which will be discussed in section 4.4.

4.1.2 WS-Security

Like WS-Addressing, WS-Security can also be configured by applying a policy to the WSDL binding.

A policy called `SecurityPolicy` is defined, and is used in the binding for the service. The name `SecurityPolicy` is arbitrary—any unique name could have been used. The `SecurityPolicy` policy has 2 policy sub-elements:

- `Wss11`, which is used to configure WS-Security 1.1 properties; and
- `AsymmetricBinding`, which specifies configuration properties for public key cryptography.

Configuring WS-Security also requires applying a policy to the WSDL messages. This policy is named `SecureMessagePolicy`. This is also an arbitrary name.

Note The `SecureMessagePolicy` policy is applied to the messages of all operations in the WSDL. The example shows how it is applied to a single operation for brevity.

An outline of the two policies is shown below.

```

<wsdl:definitions ...>
  ...
  <wsdl:binding name="DischargeSummaryReceiverBinding" type="...">
    <wsp:PolicyReference URI="#SecurityPolicy"/>
    ...
    <operation name="SendDischargeSummary">
      ...
      <input>
        ...
        <!-- Apply policy to the operation's input message -->
        <wsp:PolicyReference URI="#SecureMessagePolicy"/>
      </input>
      <output>
        ...
        <!-- Apply policy to the operation's output message -->
        <wsp:PolicyReference URI="#SecureMessagePolicy"/>
      </output>
      <fault name="InvalidIdFault">
        ...
        <!-- Apply policy to the operation's fault message -->
        <wsp:PolicyReference URI="#SecureMessagePolicy"/>
      </fault>
    </operation>
  </wsdl:binding>
  ...
  <wsp:Policy wsu:Id="SecurityPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:Wss11>
          ...
        </sp:Wss11>
        <sp:AsymmetricBinding>
          ...
        </sp:AsymmetricBinding>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

```

```

    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="SecureMessagePolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      ...
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
...
</wsdl:definitions>

```

The details of the two security policies are described in the following subsections. The following behaviour needs to be configured:

- Including timestamp;
- Signing SOAP messages;
- Encrypting SOAP messages;
- Signing before encrypting;
- Using the Basic256 algorithm suite; and
- Using specific key inclusion policies and key referencing mechanisms.

4.1.2.1 Include timestamp

Guideline TG8 recommends including a WS-Security timestamp in SOAP messages. This is configured in the security policy for the binding with the `IncludeTimestamp` element.

```

<wsdl:definitions ...>
  ...
  <wsp:Policy wsu:Id="SecurityPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        ...
        <sp:AsymmetricBinding>
          <wsp:Policy>
            ...
            <sp:IncludeTimestamp/>
            ...
          </wsp:Policy>
        </sp:AsymmetricBinding>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
  ...
</wsdl:definitions>

```

4.1.2.2 Sign SOAP messages

Guideline TG9 requires the SOAP body and headers to be signed.

The `SignedParts` element is used to indicate which parts of the SOAP messages are to be signed. This policy element must be applied individually to all operations' messages, which is why it is placed in the `SecureMessagePolicy` policy.

To conform to Guideline TG9, the body and all header elements should be enumerated within the `SignedParts` element. The timestamp should not be listed because timestamps are automatically signed.

Guideline TG9 also stipulates that the digital signatures must be calculated over the entire element (i.e. including the start and end tags of the element). The `WS-Policy OnlySignEntireHeadersAndBody` element is included to enable this behaviour. This WS-Policy element needs to be added to the security policy that applies to the WSDL binding.

```

<wsdl:definitions ...>
  ...
  <wsp:Policy wsu:Id="SecurityPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:AsymmetricBinding>
          <wsp:Policy>
            ...
            <!-- Sign elements' start and end tags,
                in addition to their contents -->
            <sp:OnlySignEntireHeadersAndBody/>
          </wsp:Policy>
        </sp:AsymmetricBinding>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

  <wsp:Policy wsu:Id="SecureMessagePolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SignedParts>
          <!-- Sign body -->
          <sp:Body/>

          <!-- Sign all WS-Addressing elements -->
          <sp:Header Namespace="http://www.w3.org/2005/08/addressing"/>

          <!-- Declarations for other header elements if any -->
          <sp:Header Name="..." Namespace="..."/>
        </sp:SignedParts>
        <sp:EncryptedParts> ...
      </sp:EncryptedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
  ...
</wsdl:definitions>

```

4.1.2.3 Encrypt SOAP messages

Guideline TG10 requires that the entire SOAP body is encrypted.

The WS-Policy `EncryptedParts` element is used to turn on encryption and to specify what parts of a message should be encrypted. It must be applied individually to all messages associated with operations. Therefore, it is placed in the `SecureMessagePolicy` policy.

Guideline TG10 also requires that only the SOAP body and the digital signature header are to be encrypted. Other headers must be passed in clear text. The SOAP body is included in the encrypted data by adding the WS-Policy `Body` element to the `EncryptedParts` element. The digital signature is included in the encrypted data by including the `EncryptSignature` element into the `SecurityPolicy` policy that applies to WSDL binding.

The default behaviour of JAX-WS is to encrypt the signature header element's start and end tags, and to avoid encrypting the `Body` element's start and end tags. This matches what Guideline TG10 requires.

```

<wsdl:definitions ...>
  ...
  <wsp:Policy wsu:Id="SecureMessagePolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SignedParts> ...
      </sp:SignedParts>
      <sp:EncryptedParts>
        <sp:Body/>
      </sp:EncryptedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>

  <wsp:Policy wsu:Id="SecurityPolicy">
    <wsp:ExactlyOne>
      <wsp:All>

```

```

    <sp:AsymmetricBinding>
      <wsp:Policy>
        ...
        <!-- Encrypt signature -->
        <sp:EncryptSignature/>
      </wsp:Policy>
    </sp:AsymmetricBinding>
    ...
  </wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>
...
</wsdl:definitions>

```

4.1.2.4 Sign before encrypting

Guideline TG11 requires SOAP messages to be signed before being encrypted. The default protection order in WS-SecurityPolicy matches this. Thus, nothing needs to be specified in the policy to sign messages before encrypting them.

Guideline TG11 says that SOAP message consumers must be able to accept messages regardless of where the timestamp appears. The order should not matter. To ensure that the position of the timestamp does not matter, specify the `Lax` layout rules in the security policy.

```

<wsdl:definitions ...>
  ...
  <wsp:Policy wsu:Id="SecurityPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        <wsp:All>
          ...
          <sp:AsymmetricBinding>
            <wsp:Policy>
              ...
              <sp:Layout>
                <wsp:Policy>
                  <sp:Lax/>
                </wsp:Policy>
              </sp:Layout>
            </wsp:Policy>
          </sp:AsymmetricBinding>
        </wsp:All>
      </wsp:ExactlyOne>
    </wsp:Policy>
  ...
</wsdl:definitions>

```

4.1.2.5 Basic256 algorithm suite

Guideline TG12 requires that the algorithms from the Basic256 algorithm suite be used. These algorithms are selected by using the `Basic256` policy element.

```

<wsdl:definitions ...>
  ...
  <wsp:Policy wsu:Id="SecurityPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        ...
        <sp:AsymmetricBinding>
          <wsp:Policy>
            ...
            <sp:AlgorithmSuite>
              <wsp:Policy>
                <sp:Basic256/>
              </wsp:Policy>
            </sp:AlgorithmSuite>
          </wsp:Policy>
        </sp:AsymmetricBinding>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
  ...
</wsdl:definitions>

```

4.1.2.6 Key inclusion policies and key referencing mechanisms

Guideline TG13 requires that the Direct Reference mechanism be used to include the service requestor's signing certificate in the request SOAP message, and the Subject Key Identifier value be used to identify the other certificates (namely the sender's encrypting, the receiver's signing and the receiver's encrypting certificates).

This behaviour is achieved by:

- Requiring support for key identifier references (using the `MustSupportRefKeyIdentifier` element);
- Specifying an *AlwaysToRecipient* inclusion policy on the initiator token (using the `IncludeToken` attribute on the `X509Token` element);
- Specifying a *Never* inclusion policy on the recipient token (using the `IncludeToken` attribute on the `X509Token` element); and
- Requiring key identifier references for references to the initiator and recipient tokens (using the `RequireKeyIdentifierReference` element).

The example fragment below shows how this behaviour is enabled in the `SecurityPolicy` policy in the WSDL file.

```
<wsdl:definitions ...>
  ...
  <wsp:Policy wsu:Id="SecurityPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:Wss11>
          <wsp:Policy>
            <!-- Require support for key identifier references -->
            <sp:MustSupportRefKeyIdentifier/>
          </wsp:Policy>
        </sp:Wss11>
        <sp:AsymmetricBinding>
          <wsp:Policy>
            <sp:InitiatorToken>
              <wsp:Policy>
                <!-- "AlwaysToRecipient" inclusion policy for initiator tokens -->
                <sp:X509Token
                  sp:IncludeToken=
"http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipient">
                  <wsp:Policy>
                    <!-- Require key identifier references for initiator
                      tokens -->
                    <sp:RequireKeyIdentifierReference/>
                    <sp:WssX509V3Token10/>
                  </wsp:Policy>
                </sp:X509Token>
              </wsp:Policy>
            </sp:InitiatorToken>
            <sp:RecipientToken>
              <wsp:Policy>
                <!-- "Never" inclusion policy for recipient tokens -->
                <sp:X509Token
                  sp:IncludeToken=
"http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never">
                  <wsp:Policy>
                    <!-- Require key identifier references for recipient
                      tokens -->
                    <sp:RequireKeyIdentifierReference/>
                    <sp:WssX509V3Token10/>
                  </wsp:Policy>
                </sp:X509Token>
              </wsp:Policy>
            </sp:RecipientToken>
            ...
          </wsp:Policy>
        </sp:AsymmetricBinding>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
  ...
</wsdl:definitions>
```

4.1.3 Specifying keys and certificates

In addition to the signing and encrypting actions, the JAX-WS toolkit will also need to know which keys and certificates to use to perform those actions. This is done through the “security callback handler” mechanism in JAX-WS.

The security callback handler is a Java class that JAX-WS invokes to obtain the keys and certificates it needs.

There are two options:

- Use the default security callback handler supplied by JAX-WS; or
- Create a custom security callback handler.

4.1.3.1 Default security callback handler

The JAX-WS toolkit provides a default security callback handler implementation in the `DefaultCallbackHandler` class. This class will be used if no security callback handler is explicitly specified in the WSDL.

The default security callback handler expects the location and identities of the keys and certificates to be configured by the WSDL file using WS-Policy statements. The client’s key is selected using the `KeyStore` element, and the service’s certificate using the `TrustStore` element. These elements belong to a proprietary Sun namespace; they are specific to the JAX-WS toolkit.

```
<definitions ...>
...
<wsdl:binding name="DischargeSummaryReceiverBinding" type="...">
  <wsp:PolicyReference URI="#SecurityConfigPolicy"/>
  ...
</wsdl:binding>
...
<wsp:Policy wsu:Id="SecurityConfigPolicy">
  <wsp:ExactlyOne>
    <wsp:All xmlns:sc="http://schemas.sun.com/2006/03/wss/client">
      <sc:KeyStore alias="client_cert" storetype="jks"
        location="C:\\certs\\keystore.jks" storepass="password"/>
      <sc:TrustStore peeralias="server_cert" storetype="jks"
        location="C:\\certs\\truststore.jks" storepass="password"/>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
...
</definitions>
```

The limitation of this approach is that this information is hard-coded into the WSDL. This is not practical if the client needs to invoke multiple service instances that use different security certificates. A separate WSDL would be required for each service instance since the WSDL sets the service’s certificate to use in this approach.

4.1.3.2 Custom security callback handler

With a custom security callback handler, the developer has complete control over what keys and certificates are used and how they are obtained. The developer creates a Java class and JAX-WS uses it as its security callback handler.

This is the approach that will be used in this example implementation.

JAX-WS is configured to use the custom Java class with the `CallbackHandlerConfiguration` policy element in the WSDL. The actual implementation of the Java class will be described in section 4.4. The `CallbackHandlerConfiguration` element belongs to a proprietary Sun namespace; it is specific to the JAX-WS toolkit.

```

<definitions ...>
  ...
  <wsdl:binding name="DischargeSummaryReceiverBinding" type="...">
    <wsp:PolicyReference URI="#SecurityConfigPolicy"/>
    ...
  </wsdl:binding>
  ...
  <wsp:Policy wsu:Id="SecurityConfigPolicy">
    <wsp:ExactlyOne>
      <wsp:All xmlns:sc="http://schemas.sun.com/2006/03/wss/client">
        <sc:CallbackHandlerConfiguration>
          <sc:CallbackHandler name="xwssCallbackHandler"
            classname="com.example.common.security.SecurityCallbackHandler" />
        </sc:CallbackHandlerConfiguration>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
  ...
</definitions>

```

4.2 Generate the service interface classes from the WSDL files

The service interface classes are generated by running the *wsimport* tool over the WSDL [WSIMPORT]. The *wsimport* tool comes as a command-line application as well as an Ant task. Ant is a Java build tool. The examples below are based on calling *wsimport* from Ant.

The *wsimport* Ant task must be declared before using it. The Java class for this Ant task comes with the JAX-WS distribution. The code below assumes that there is a `JAXWS_HOME` environment variable that points to the root directory of the JAX-WS toolkit.

```

<!-- Define path to JAX-WS JAR files -->
<property environment="env"/>
<path id="jaxws.path">
  <fileset dir="${env.JAXWS_HOME}/lib" includes="*.jar"/>
</path>

<!-- Define wsimport task -->
<taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport">
  <classpath refid="jaxws.path"/>
</taskdef>

```

The following are commonly useful parameters that should be passed to *wsimport*:

- `wsdl`: WSDL file from which to generate classes;
 - This file is the WSDL containing the service instance information.
- `wsdllocation`: value that will be in the `@WebServiceClient.wsdlLocation` annotation in the generated service interface;
- `package`: package name that the generated classes should be in;
- `destdir`: location to place the generated compiled classes;
- `keep`: set to true to keep the source files of the generated classes;
- `sourcedestdir`: location to place the generated source files;
- `extension`: set to true to enable custom extensions; and
 - The JAX-WS toolkit regards the SOAP 1.2 binding as an extension. Thus, this parameter must be set to true to use SOAP 1.2, which is recommended by the NEHTA *Web Services Standards Profile* [WSSP2006].
- `binding`: location of the JAXB binding file.

A JAXB binding file is necessary to declare SOAP handlers in the example implementation. This binding file will be discussed next

```
<wsimport wsdl="wsdl/DischargeSummaryReceiver.wsdl"
  wsdllocation="DischargeSummaryReceiver.wsdl"
  package="com.example.ds.client"
  destdir="gen/classes"
  keep="true"
  sourcedestdir="gen/src"
  extension="true"
  binding="conf/handlers.xml"/>
```

A JAXB binding file can be specified in the call to the *wsimport* tool. This binding file, whose contents are shown below, should specify a handler chain that enumerates the handler classes that apply to our client. Section 4.4 will discuss what SOAP handlers are, how to implement them, and how to code handler to make SOAP messages conform to the Guidelines.

```
<bindings xmlns="http://java.sun.com/xml/ns/jaxws"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  wsdlLocation="wsdl/DischargeSummaryReceiver.wsdl">
  <bindings node="wsdl:definitions">
    <jws:handler-chains xmlns:jws="http://java.sun.com/xml/ns/javaee">
      <jws:handler-chain>
        <jws:handler>
          <jws:handler-class>
            com.example.common.addressing.client.AddressingHandler
          </jws:handler-class>
        </jws:handler>
      </jws:handler-chain>
    </jws:handler-chains>
  </bindings>
</bindings>
```

4.3 Implement the client application

The steps for a client to invoke a Web service operation are:

- Create a service object;
- Get a port object from the service object;
- Set the request context properties on the port object; and
- Invoke Web service operations on the port object.

```
package com.example.ds.client;
import java.io.FileInputStream;
import java.io.InputStream;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.URL;
import java.util.Map;
import java.util.Properties;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.WebServiceClient;
import com.example.common.client.ClientProperty;
import com.example.common.util.ClasspathUtil;
public class DSSimpleClient {
  public static void main(String[] args) throws Exception {
    // Create service
    DischargeSummaryReceiverService service = createService();

    // Create port
    DischargeSummaryReceiver port = service.getDischargeSummaryReceiver();

    // Set request context properties on the port
    Properties configProperties = ...; // Read in config properties
    setRequestContextProperties(port, configProperties);

    // Invoke Web service operations
    ReceivedStatusType status = port.checkStatus("1234");
  }
  ...
}
```

4.3.1 Create a service object

The *wsimport* tool generates a class that corresponds to each service declaration in the WSDL, which this document will call a “service class”. In the example implementation, this service class is named `DischargeSummaryReceiverService`. The first step in invoking Web service calls is to create an instance of this service class.

The generated service class has a constructor that has no parameters. It is advisable not to call this constructor when instantiating a new service object. It uses a hard-coded value for the WSDL location, which corresponds to the location of the WSDL file when the classes were generated by the *wsimport* tool. Calling this constructor when a local WSDL file is used to generate the classes will mean that the client application is not portable. If a remote WSDL location is used to generate the classes, the client application will stop working if the remote WSDL location is changed (e.g. the WSDL file is moved to a new remote location).

The generated service class also has a constructor that takes a WSDL location. This constructor should be used since it gives the flexibility to choose the location of the WSDL file at run-time. The WSDL at the specified location should contain the policy information specified in section 4.1. The constructor also requires passing the qualified name of the service in the WSDL. Instead of using hard-coded values, the namespace and the name of the service can be retrieved from the `WebServiceClient` annotation in the generated service class.

```
public static void main(String[] args) throws Exception {
    // Create service
    DischargeSummaryReceiverService service = createService();
    ...
}

private static DischargeSummaryReceiverService createService() {
    URL wsdlLoc = ...; // Locate WSDL
    QName serviceQName = new QName(getServiceNamespace(), getServiceName());
    return new DischargeSummaryReceiverService(wsdlLoc, serviceQName);
}

private static String getServiceNamespace() {
    WebServiceClient annotation = (WebServiceClient)
        DischargeSummaryReceiverService.class.getAnnotation(WebServiceClient.class);
    return annotation.targetNamespace();
}

private static String getServiceName() {
    WebServiceClient annotation = (WebServiceClient)
        DischargeSummaryReceiverService.class.getAnnotation(WebServiceClient.class);
    return annotation.name();
}
```

4.3.2 Get a port object

The service class has a `get()` method for each port declared in the WSDL. These `get()` methods return a port object that allows the client application to invoke operations defined for that port in the WSDL.

```
public static void main(String[] args) throws Exception {
    ...
    // Get port
    DischargeSummaryReceiver port = service.getDischargeSummaryReceiver();
    ...
}
```

4.3.3 Set request context properties on the port

The request context properties allow the client’s main application to specify values that are used by other parts of the program.

The client application should set the `BindingProvider.ENDPOINT_ADDRESS_PROPERTY` property in the request context. This is a built-in property of the JAX-WS toolkit. When it is set, its value will override the value of the endpoint address in the WSDL. This feature enables the client to dynamically select the service that it wishes to invoke.

In order for the client to dynamically select a Web service, it needs to also be able to dynamically select the service's certificate, which is used to encrypt requests. This is also done using request context properties, which are made available to the security callback handler and in turn the cert selector classes. Section 4.4 will explain what these security classes are and how to implement them. The client application passes the Subject name of the certificate to select in a request context property to these security classes. This is a custom request context property.

The request context properties set by the client application can also be accessed by SOAP handlers. A SOAP handler is necessary in the example implementation to set the WS-Addressing To and From headers to the service's and client's logical identifiers respectively. Section 4.4 explains how to implement this SOAP handler. The client application can pass the values of the service's and client's logical identifiers to this SOAP handler using request context properties. These are also custom request context properties.

```
public static void main(String[] args) throws Exception {
    ...
    // Set request context properties on the port
    Properties configProperties = ...; // Read in config properties
    setRequestContextProperties(port, configProperties);
    ...
}

private static void setRequestContextProperties(
    DischargeSummaryReceiver port, Properties configProperties)
    throws URISyntaxException {
    // Get request context properties
    BindingProvider bindingProvider = (BindingProvider) port;
    Map<String, Object> requestContext = bindingProvider.getRequestContext();

    // Set endpoint address URL
    String endpointAddress = configProperties.getProperty("service.url");
    requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
        endpointAddress);

    // Set service's certificate
    String serviceCertificate = configProperties
        .getProperty("service.certSubject");
    requestContext.put(ClientProperty.SERVICE_CERT, serviceCertificate);

    // Set service's logical name
    String serviceLogicalName = configProperties
        .getProperty("service.logicalName");
    requestContext.put(ClientProperty.WSA_TO, new URI("urn:dsr:"
        + serviceLogicalName));

    // Set client's logical name
    String clientLogicalName = configProperties
        .getProperty("client.logicalName");
    requestContext.put(ClientProperty.WSA_FROM, new URI("urn:dsr:"
        + clientLogicalName));
}
```

The custom request context properties can have arbitrary names, as long as they are agreed upon between the client's main application and the handlers. In addition, the custom property names should not clash with JAX-WS property names. The names of the request context properties can be specified as constants in an interface.

```
package com.example.common.client;
public interface ClientProperty {
    String SERVICE_CERT = "com.example.common.security.certificate.service";
    String WSA_TO = "com.example.common.addressing.to";
    String WSA_FROM = "com.example.common.addressing.from";
}
```

4.3.4 Invoke Web service operations

The client application invokes Web service operations on the port object.

Although the example WSDL defined in section 2.3.2.1 uses the document-literal style, since it follows the wrapped convention, the *wsimport* tool recognises this convention and generates RPC style method signatures.

```
public static void main(String[] args) throws Exception {
    ...
    // Invoke Web service operations
    ReceivedStatus status = port.checkStatus("1234");
}
```

4.4 Implement a SOAP handler

Guideline TG14 recommends providing the WS-Addressing *To*, *From*, *Action* and *MessageID* headers in SOAP request messages. In addition, the *To* and *From* headers should contain logical identifiers for the service and the client respectively.

To conform to Guideline TG14, this example implementation has to edit the WS-Addressing headers in the request messages. It needs to:

- Set the *To* header to the service's logical identifier (by default, it is set to the service's URL);
- Add the *From* header (by default, it is not provided in request messages); and
- Remove the *ReplyTo* header (by default, it is provided in request messages, but is not part of the Guideline).

The above changes to requests can be done within a SOAP handler. SOAP handlers are a programmatic mechanism that provides access to SOAP messages before they are sent and after they are received [HANDLER]. In this example implementation, a SOAP handler called *AddressingHandler* will be used to edit WS-Addressing headers to conform to Guideline TG14.

The code for this SOAP handler is given below. Basically, a SOAP handler must implement the *SOAPHandler<SOAPMessageContext>* interface. This interface has a *handleMessage()* method, which gets passed a *SOAPMessageContext* object. The *getMessage()* method on this context object returns the SOAP message that is to be sent or has just been received. The

The *AddressingHandler* SOAP handler first checks that the SOAP message is an outbound message since we are only concerned with editing the WS-Addressing headers on client requests. This is done by retrieving the *MessageContext.MESSAGE_OUTBOUND_PROPERTY* Boolean property from the *SOAPMessageContext* object.

The *SOAPMessageContext* object also provides access to request context properties that are set by the client's main application. Section 4.3.3 explained how to set these properties. This mechanism basically allows the client application to pass the *To* and *From* values to the SOAP handler.

The SOAP message can then be manipulated using a SOAP object model API that extends the DOM API.

```
package com.example.common.addressing.client;

import java.util.Iterator;
import java.util.Set;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
```

```

import javax.xml.ws.handler.soap.SOAPMessageContext;
import com.example.common.client.ClientProperty;

public class AddressingHandler implements SOAPHandler<SOAPMessageContext> {
    private static final String WSA_PREFIX = "wsa";
    private static final String WSA_NS = "http://www.w3.org/2005/08/addressing";

    public boolean handleMessage(SOAPMessageContext context) {
        try {
            SOAPMessage message = context.getMessage();
            SOAPHeader header = message.getSOAPHeader();

            if (isOutboundMessage(context) && (header != null)) {
                // Set "To" header to a logical name
                String toValue = (String) context.get(ClientProperty.WSA_TO);
                SOAPElement toElem = getFirstChild(header, "To", WSA_NS);
                if (toElem != null) {
                    toElem.setTextContent(toValue);
                }

                // Add "From" header
                String fromValue = (String) context.get(ClientProperty.WSA_FROM);
                SOAPElement fromElem = header.addChildElement("From", WSA_PREFIX,
                    WSA_NS);
                SOAPElement fromAddrElem = fromElem.addChildElement("Address",
                    WSA_PREFIX, WSA_NS);
                fromAddrElem.setTextContent(fromValue);

                // Remove "ReplyTo" header
                SOAPElement replyToElem = getFirstChild(header, "ReplyTo", WSA_NS);
                if (replyToElem != null) {
                    header.removeChild(replyToElem);
                }
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        // Continue processing
        return true;
    }

    ... // Empty implementations of other methods of SOAPHandler

    private static boolean isOutboundMessage(SOAPMessageContext context) {
        Boolean result = (Boolean) context
            .get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
        if (result == null) {
            String msg = "MessageContext.MESSAGE_OUTBOUND_PROPERTY property was "
                + "not set.";
            throw new RuntimeException(msg);
        }
        return result.booleanValue();
    }

    private static SOAPElement getFirstChild(SOAPHeader header, String name,
        String ns) {
        SOAPElement result = null;
        QName qname = new QName(ns, name);
        Iterator iter = header.getChildElements(qname);
        if (iter.hasNext()) {
            result = (SOAPElement) iter.next();
        }
        return result;
    }
}

```

4.5 Implement the security callback handler

This step provides the implementation of the custom security callback handler. This is the Java class that was referenced by the WSDL file, as described in section 4.1.3.2.

The role of the security callback handler is to obtain security-related information, such as username-password pairs, private keys and certificates. It also provides the implementation of certain security tasks, such as checking

timestamps and validating certificates. It does not carry out major security tasks, such as signing and encrypting.

For this example implementation, the custom security callback handler behaves the same as the default security callback handler except that it can be configured at run-time. It is implemented as a wrapper around the default `DefaultCallbackHandler` class.

All security callback handler classes must implement the `CallbackHandler` interface, which has one method called `handle()`. This method is called when the JAX-WS security framework needs the security callback handler to carry out a particular task, such as locate the key to sign SOAP messages. In the custom security callback handler, all calls on the `handle()` method is delegated to the `DefaultCallbackHandler` instance.

The constructor of the custom security callback handler creates an instance of the `DefaultCallbackHandler`, configuring it with a `Properties` object. The configuration properties can be loaded in several ways, such as from a file or from a database. For simplicity, the example implementation uses the `System` properties.

```
package com.example.common.security;

import java.io.IOException;
import java.util.Properties;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import com.sun.xml.wss.XWSSecurityException;
import com.sun.xml.wss.impl.misc.DefaultCallbackHandler;

public class SecurityCallbackHandler implements CallbackHandler {

    private final DefaultCallbackHandler defaultHandler;

    public SecurityCallbackHandler() throws XWSSecurityException {
        // Get configuration properties for default handler in some custom way
        // For code simplicity, example implementation retrieves
        // DefaultCallbackHandler's config properties from System properties
        Properties configProps = System.getProperties();

        // Creates a DefaultCallbackHandler instance passing in the configuration
        // properties
        this.defaultHandler = new DefaultCallbackHandler("", configProps);
    }

    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {

        // Delegate to the default handler
        this.defaultHandler.handle(callbacks);
    }
}
```

The `DefaultCallbackHandler` class has several configuration properties, but not all of them have to be set. This section will describe only those properties that should be set.

Note: The properties that will be described are only relevant because this custom security callback handler is a wrapper around the `DefaultCallbackHandler` class. A custom security callback handler that is not based upon the `DefaultCallbackHandler` does not need to use these properties.

The `DefaultCallbackHandler` instance has to return the certificates and private keys to use for signing and encrypting requests and then verifying and decrypting responses. In order to do this, it needs to be able to locate and open the key store and trust store. A key store is a security key database file that contains the certificate and private key for the entity. In this case, the entity is the client. A trust store is a security key database file that contains the public certificates of others, such as the certificate authority, Web

services. *Appendix C*: Key management contains more information on these files. The following configuration properties must be set since they are used by the `DefaultCallbackHandler` to locate and open the key store and trust store:

- `keystore.url`: file location of the key store;
- `keystore.type`: type of the key store (*pkcs12*, *jks* or *jceks*);
- `keystore.password`: password for the key store;
- `truststore.url`: file location of the trust store;
- `truststore.type`: type of the trust store (*pkcs12*, *jks* or *jceks*); and
- `truststore.password`: password for the trust store.

To encrypt requests, the `DefaultCallbackHandler` instance needs to return the certificate of the Web service that will be invoked. We can select this certificate by setting the `peerentity.alias` configuration property to the alias for the certificate in the trust store. However, this approach will tie the client to a particular service when the client is running. To change the service's certificate will involve closing the client, editing the configuration, and restarting the client.

A more flexible approach is to provide a custom trust store `CertSelector` class. To implement this approach, the `truststore.certselector` configuration property to the `DefaultCallbackHandler` should be set to the fully qualified name of the custom trust store `CertSelector` class.

A sample implementation of the trust store `CertSelector` class is provided below. It implements the `CertSelector` interface. It should have a constructor that takes in a `Map` object. This `Map` object is passed by the `DefaultCallbackHandler` and contains the request context properties set by the client's main application. Section 4.3.3 discussed how these request context properties are set. One of these properties should be the Subject name of the certificate for the Web service that the client's main application will invoke.

The `CertSelector` interface has two methods to implement: `match()` and `clone()`. The `match()` method is called as the `DefaultCallbackHandler` iterates through the certificates in the trust store. This method should return true when it is passed the certificate to select – namely the certificate whose Subject name matches the value of the client's request context property. The `clone()` method implementation provides a duplicate copy of the `CertSelector`. The `@Override` annotation marks that this method implementation overrides the protected `Object.clone()` method.

```
package com.example.common.security.client;

import java.security.Principal;
import java.security.cert.CertSelector;
import java.security.cert.Certificate;
import java.security.cert.X509Certificate;
import java.util.HashMap;
import java.util.Map;
import com.example.common.client.ClientProperty;

public class ClientTrustStoreCertSelector implements CertSelector {
    private Map context;
    private String serviceCert;

    public ClientTrustStoreCertSelector(Map contextParam) {
        this.context = contextParam;
        if (context != null) {
            this.serviceCert = (String) context.get(ClientProperty.SERVICE_CERT);
        }
    }

    public boolean match(Certificate certificate) {
        boolean result = false;
    }
}
```

```
    if ((this.serviceCert != null) && (this.serviceCert.length() > 0)
        && (certificate instanceof X509Certificate)) {
        // The certificate's subject name must match what has been specified
        String certSubj = getCertificateSubject((X509Certificate) certificate);
        result = this.serviceCert.equals(certSubj);
    }
    return result;
}

@Override
public Object clone() {
    return new ClientTrustStoreCertSelector(new HashMap(this.context));
}

private String getCertificateSubject(X509Certificate certificate) {
    String result = null;
    Principal subject = certificate.getSubjectDN();
    if (subject != null) {
        result = subject.getName();
    }
    return result;
}
}
```

To sign requests, the `DefaultCallbackHandler` selects by default the first entry in the key store with a private key. This behaviour is acceptable for the general case when the client has a single pair of public certificate and private key associated with it. If this behaviour is incorrect, set the `my.alias` configuration property to the alias of the key to use in the key store or set the `keystore.certselector` property to a cert selector class that selects the key in a custom-defined way. In addition, the `DefaultCallbackHandler` uses by default the same password for the private key as that of the key store. If the password for the private key is different from the key store, set the `key.password` configuration property.

No additional configuration properties need to be set to verify and decrypt responses. The `DefaultCallbackHandler` can automatically find the certificates and private keys in the key store and trust store from the key references in the response messages. For example, to decrypt a response, the Subject Key Identifier value of the encryption key in the response, which is the client's public key, is used to look up the matching private key in the key store.

5 Web service

This chapter describes how to build a Web service using JAX-WS. The aim of a Web service is to provide a service instance that makes available service operations for clients to invoke.

The steps for building a Web service are:

1. Specify the policies in the WSDL;
2. Generate the service interface classes from the WSDL files;
3. Implement the Web service;
4. Implement the security callback handler;
5. Configure the Web service;
6. Package the Web service; and
7. Deploy the Web service.

In some steps, there are several similarities with the steps in building a Web service client. For these steps, this section is written describing what additional actions or modifications need to be done on the step described on the client-side to adapt it to the server-side.

5.1 Specify the policies in the WSDL

The policies in the WSDL for the client, described in section 4.1, are almost the same for the server-side. The policies for specifying WS-Addressing and WS-Security (namely `AddressingPolicy`, `SecurityPolicy` and `SecureMessagePolicy`) are exactly the same.

The only difference is in the policy for specifying keys and certificates, `SecurityConfigPolicy`. The XML namespace for setting the security callback handler is different for the server. The XML elements, e.g. `CallbackHandlerConfiguration` and `CallbackHandler`, belong to `http://schemas.sun.com/2006/03/wss/server` namespace on the server-side. On the client-side, they belong to the `http://schemas.sun.com/2006/03/wss/client` namespace. The same security callback handler class can be used on the server-side as the client-side.

```
<definitions ...>
  ...
  <wsp:Policy wsu:Id="SecurityConfigPolicy">
    <wsp:ExactlyOne>
      <wsp:All xmlns:sc="http://schemas.sun.com/2006/03/wss/server">
        <sc:CallbackHandlerConfiguration>
          <sc:CallbackHandler name="xwssCallbackHandler"
            classname="com.example.common.security.SecurityCallbackHandler"/>
        </sc:CallbackHandlerConfiguration>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
</definitions>
```

5.2 Generate the service interface classes from the WSDL files

The `wsimport` tool is also used to generate the service interface classes for the server. Most of the information for this step in the client-side, explained in section 4.2, also applies to the server-side.

The exception is that there is no need for SOAP handlers on the server-side. Thus, a JAXB binding file is not needed and the `binding` attribute should be removed from the call to the `wsimport` Ant task.

5.3 Implement the Web service

A Web service implementation class provides concrete method implementations for the operations in the WSDL. Although it is not necessary, it is a good idea for the Web service implementation class to implement the generated port interface, which is the `DischargeSummaryReceiver` interface in this example. Using this approach, some discrepancies, such as missing method implementations, can be discovered at compile-time.

The Web service implementation class must have a `WebService` annotation with an `endpointInterface` attribute. The value of this `endpointInterface` attribute must be the fully-qualified name of the generated port interface. The `WebMethod` annotation must be used to mark the methods that implement the operations in the WSDL.

```
package com.example.ds.server;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService (endpointInterface="com.example.ds.server.DischargeSummaryReceiver")
public class DischargeSummaryReceiverImpl implements DischargeSummaryReceiver {
    @WebMethod
    public void ping() {
        ...
    }

    @WebMethod
    public void sendDischargeSummary(DischargeSummaryType document)
        throws InvalidIdFault_Exception {
        ...
    }

    @WebMethod
    public ReceivedStatusType checkStatus(String documentId)
        throws InvalidIdFault_Exception {
        ...
    }
}
```

5.4 Implement the security callback handler

Section 4.2 discussed how to implement the security callback handler on the client-side. The same security callback handler class can be used on the server-side. Values should be provided for the properties to locate and open the key store and trust store, namely: `keystore.url`, `keystore.type`, `keystore.password`, `truststore.url`, `truststore.type` and `truststore.password`.

Unlike the client-side configuration, a custom trust store `CertSelector` is not always necessary. By default, the JAX-WS implementation will use the client's certificate in the request message to encrypt the response message. Generally, this default behaviour is appropriate, but there are cases when it will have to be changed. For example, if the client's certificate in the request only permits *Digital Signature* usage, it cannot be used to encrypt the response. In that case, a custom trust store `CertSelector` is necessary to look up the client's certificate that permits encryption.

5.5 Package the Web service

5.5.1 Directory structure

A Web service is a Web application. The directory structure of a Web service, which is given below, follows the standard structure of a Web application.

Since there is no visible aspect to a Web service, all files are located within the *WEB-INF* directory. The *web.xml* and *sun-jaxws.xml* configuration files, which are discussed in the section 0, must be in this directory.

All Java classes, namely the generated stubs, the Web service implementation class and other classes, must be in the *classes* sub-directory.

Any JAR files used by the Web service implementation must be in the *lib* sub-directory. The JAX-WS JAR files should not be included.

There is no prescribed location for the WSDL and XSD files of a Web service. JAX-WS locates the service's WSDL from what is specified in the *sun-jaxws.xml* configuration file. The WSDL and XSD files should be placed in a single directory to create a clean directory structure for the Web service.

```

/WEB-INF
  web.xml
  sun-jaxws.xml
  classes/
    (Java class files)
  lib/
    (JAR files)
  wsdl/
    (WSDL and XSD files)

```

5.5.2 Create the configuration files

There are two configuration files needed for the Web service: *web.xml* and *sun-jaxws.xml*.

5.5.2.1 web.xml

Since the Web service is a Web application, it requires a *web.xml* deployment descriptor. An example *web.xml* file is provided below.

The *web.xml* file specifies the servlet that will handle the HTTP requests, `com.sun.xml.ws.transport.http.servlet.WSServlet`, and its context listener, `com.sun.xml.ws.transport.http.servlet.WSServletContextListener`. These classes are part of the JAX-WS toolkit.

```

<web-app>
  <listener>
    <listener-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>
  <servlet>
    <servlet-name>DischargeSummaryReceiver</servlet-name>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>DischargeSummaryReceiver</servlet-name>
    <url-pattern>*/</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>

```

5.5.2.2 sun-jaxws.xml

A *sun-jaxws.xml* file configures the Web service's endpoints in JAX-WS. An example configuration file is provided below.

```
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
  <endpoint name="DischargeSummaryReceiver"
    implementation="com.example.dischargesummaryreceiver.server.DischargeSummaryReceiverImpl"
    wsdl="WEB-INF/wsdl/DischargeSummaryReceiverService.wsdl"
    service="{http://example.com/ns/2007/DischargeSummaryReceiver/v1}DischargeSummaryReceiverService"
    port="{http://example.com/ns/2007/DischargeSummaryReceiver/v1}DischargeSummaryReceiver"
    url-pattern="/*"
    binding="http://java.sun.com/xml/ns/jaxws/2003/05/soap/bindings/HTTP/"
  />
</endpoints>
```

A Web service endpoint is declared using an `endpoint` XML element. The following XML attributes should be provided in the endpoint element:

- *name*: descriptive name of the endpoint;
- *implementation*: qualified name of the Web service implementation class written in section 5.3;
- *wsdl*: location of the root WSDL;

The root WSDL is the WSDL containing the service declaration. The example assumes the directory structure described in section 5.5.1. All WSDL and XSD files are in the *WEB-INF/wsdl* directory.

- *service*: qualified name of the service;

The namespace should match WSDL's target namespace. The local name should match the service's name defined in the WSDL.
- *port*: qualified name of the port;

The namespace should match WSDL's target namespace. The local name should match the port's name defined in the WSDL.
- *url-pattern*: specifies which URLs will map to this endpoint; and
- *binding*: which SOAP binding to use.

The SOAP 1.2 binding should be used in order to conform to the NEHTA *Web Services Standards Profile* [WSSP2006]. The value for the SOAP 1.2 binding is:
`http://java.sun.com/xml/ns/jaxws/2003/05/soap/bindings/HTTP/`

5.5.3 Create the WAR file

Since the Web service is a Web application, it must be packaged in a Web Archive (WAR) file. The WAR file can be created using the *war* Ant task or the *jar* command-line tool. An example call to the *war* Ant task is given below.

```
<war warfile="dsreceiver.war" webxml="${conf.dir}/web.xml">
  <webinf dir="${conf.dir}">
    <include name="sun-jaxws.xml" />
  </webinf>
  <classes dir="${classes.dir}">
    <include name="**/*.class" />
  </classes>
  <zipfileset dir="${wsdl.dir}" prefix="WEB-INF/wsdl">
    <include name="*.wsdl" />
    <include name="*.xsd" />
  </zipfileset>
</war>
```

5.6 Deploy the Web service

5.6.1 Install JAX-WS on the server

Appendix B: Installation contains instructions and notes on installing JAX-WS on the server. It has specific details on the Sun Java System Application Server (SJSAS), which is Sun's J2EE reference implementation.

5.6.2 Configure the server for the Web service

The custom security callback handler, which is described in section 4.4 and 5.4, reads its configuration details from System properties. The following properties will need to be specified: `keystore.url`, `keystore.type`, `keystore.password`, `truststore.url`, `truststore.type` and `truststore.password`.

How System properties are specified on the server will differ between server software. *Appendix section B.6.1* explains how to specify System properties in SJSAS.

5.6.3 Deploy the WAR file

Deploying the Web service will depend on the chosen server software. Read the server's user guide on how to deploy a Web application.

Several Java servers have an auto-deploy or hot deploy feature. The server will automatically deploy WAR files copied to a particular directory. For SJSAS, this directory is: `$AS_HOME/domains/domain1/autodeploy`. Copy the WAR file created in section 5.5.3 to the auto-deploy directory.

5.6.4 Accessing the Web service

The Web service should be available at: `http://<host>:<port>/<application name>`, where the `<application name>` matches the prefix of the WAR file. For example, if the `dsreceiver.war` file was deployed on a server XYZ running on port 8080, the URL would be `http://XYZ:8080/dsreceiver`.

The Web service's WSDL is available by appending a `"?wsdl"` on the service's URL, e.g. `http://XYZ:8080/dsreceiver?wsdl`. In the published WSDL, the server will automatically replace the dummy value for the address in the original WSDL (namely `"http://DUMMY_VALUE"`) with the actual address of the service.

Appendix A: References

- [EJBCA] EJBCA, EJBCA – The J2EE Certificate Authority, <http://ejbca.sourceforge.net>.
- [GIIWS2007] NEHTA, Guidelines for Implementing Interoperable Web Services, version 1.0, 28 March 2007.
- [HANDLER] Sun Microsystems, Java API for XML Web Services (JAX-WS) Handler, version 2.1, revision 1.1, <https://jax-ws.dev.java.net/nonav/2.1/docs/handlers.html>.
- [JCA] Sun Microsystems, Java Cryptography Architecture (JCA), <http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html>.
- [JAXWS] Sun Microsystems, Java API for XML Web Services (JAX-WS) Handler, <https://jax-ws.dev.java.net>.
- [JSR224] Java API for XML-Based Web Services, 2.0, JSR-000224, <http://jcp.org/aboutJava/communityprocess/pfd/jsr224/>.
- [KEYTOOL] Sun Microsystems, keytool-Key and Certificate Management Tool, <http://java.sun.com/javase/6/docs/technotes/tools/windows/keytool.html>.
- [NDS2006] NEHTA, National Discharge Summary: Data Content Specifications, version 1.0, 21 December 2006.
- [NIF2006] NEHTA, Interoperability Framework, version 1.0, 1 April 2006.
- [OPENSSL] OpenSSL, OpenSSL: The Open Source toolkit for SSL/TLS, <http://www.openssl.org>.
- [PING] Muus, The Story of the PING Program, <http://ftp.arl.mil/~mike/ping.html>.
- [PKCS1999] RSA Laboratories, PKCS 12: Personal Information Exchange Syntax, version 1.0, 24 June 1999, <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-12/pkcs-12v1.pdf>.
- [TAIS2006] NEHTA, Technical Architecture for Implementing Services: Concepts and Patterns, version 1.0, 21 December 2006.
- [WCF] Microsoft, Windows Communication Foundation (WCF), <http://wcf.netfx3.com>.
- [WSE] Microsoft, Web Services Enhancements (WSE), <http://msdn2.microsoft.com/en-us/webservices/aa740663.aspx>.
- [WSIMPORT] Sun Microsystems, Java API for XML Web Services (JAX-WS) wsimport, version 2.1, revision 1.1, <https://jax-ws.dev.java.net/nonav/2.1/docs/wsimport.html>.
- [WSSP2006] NEHTA, Web Services Standards Profile, version 2.0, 20 November 2006.

Appendix B: Installation

B.1 Java Development Kit

1. Download and install JDK 5 or later.

URL: <http://java.sun.com/javase/downloads/index.jsp>

<JDK_HOME> will be used in this document to refer to the root directory of the JDK installation.

<JRE_HOME> will be used in this document to refer to <JDK_HOME>/jre.

2. Add <JDK_HOME>/bin to the path.
3. Create a JAVA_HOME environment variable pointing to <JRE_HOME>.

B.2 Java Cryptography Extension Unlimited Strength Jurisdiction Policy Files

The cryptography in the JDK download is limited in strength due to the import control restrictions for some countries. The “unlimited strength” capabilities are enabled by installing certain policy files into the JRE.

1. Download the JCE Unlimited Strength Jurisdiction Policy Files for the installed JDK version.

URL: <http://java.sun.com/javase/downloads/index.jsp>

2. Unpack the downloaded zip file.
3. Copy the two JAR files (*local_policy.jar* and *US_export_policy.jar*) to the <JRE_HOME>/lib/security directory.

Overwrite the existing JAR files in the directory.

B.3 Ant

Ant is a Java-based build tool. JAX-WS comes with Ant scripts to simplify its installation. To use these scripts, the Ant tool must be installed.

1. Download Ant 1.6.5 or later.

URL: <http://ant.apache.org/bindownload.cgi>

2. Unpack the downloaded zip file to the desired location.

<ANT_HOME> will be used in this document to refer to the root directory of Ant.

3. Add <ANT_HOME>/bin to the path.

B.4 JAX-WS

1. Download *JAX-WS 2.1 with WSIT Milestone Release 4*.

URL: <https://jax-ws.dev.java.net/servlets/ProjectDocumentList?folderID=5648>

2. Unpack the downloaded JAR file with the command: `java -jar wsit-1_0-fcs-bin-b14-09_apr_2007.jar`

A dialog box for the license agreement will appear. Read this license agreement. The *Accept* button will only be enabled when the scroll bar reaches the bottom.

If you accept the license agreement, the files will be unpacked to a *jax-ws-latest-wsit* sub-directory within the directory containing the JAX-WS JAR file.

3. Move the *jax-ws-latest-wsit* directory to the desired location.

`<JAXWS_HOME>` will be used in this document to refer to this *jax-ws-latest-wsit* directory.

JDK 6 comes with JAX-WS 2.0, which is an older version of JAX-WS that lacks some functionality necessary to comply with the guidelines, such as support for WS-Addressing. This older JAX-WS version needs to be overridden using the endorsed directory mechanism of Java. The JAX-WS installation comes with an Ant script that ensures the JRE uses its version of JAX-WS instead of the old version that the JRE has.

1. If JDK 6 is installed, run in the `<JAXWS_HOME>` directory: `ant -f wsit-on-glassfish.xml install-api`

This command copies the *webservices-api.jar* file from `<JAXWS_HOME>/lib` directory to the `<JRE_HOME>/lib/endorsed`.

B.5 Servlet container

This section describes in general terms setting up a servlet container to host a JAX-WS Web service. How these steps are carried out will depend on the server software.

1. Install the server software.
2. Install *JAX-WS 2.1 with WSIT* on the server.

This step involves making the server load the JAR files in the *lib* directory of the JAX-WS installation. For instance, for some server software, this step involves copying the JAX-WS JAR files to the server's own *lib* directory.

3. Turn off capture stack trace feature of JAX-WS by setting the following System property: `com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace=false`

If the capture stack trace feature is turned on, which it is by default, when a fault or exception occurs, JAX-WS will place the stack trace of the fault or exception in the `Details` element of the SOAP fault message that is returned to the client. This feature is undesirable for two reasons.

It exposes the Web service to unknown clients.

It causes interoperability issues with Microsoft WCF clients. WCF expects only one child element within the `Details` element of a SOAP fault. When a fault is raised and the capture stack trace feature is turned on, JAX-WS will place 2 child elements within the `Details` element: one for the fault (whose structure is defined in the WSDL) and another for the stack trace. WCF clients then cannot recognise the fault that was returned. Another exception about the SOAP fault being invalid will be raised on the client.

B.6 Sun Java System Application Server (SJSAS)

This section explains how to do the steps described in the previous section in terms of SJSAS, Sun's J2EE reference implementation.

To install the server software:

1. Download SJSAS 9 or later.

URL: <http://java.sun.com/javaee/downloads/index.jsp>

2. Run the downloaded file by double-clicking on it.

A wizard should step you through the installation process, such as selecting the directory to install in and the ports to use.

<SJSAS_HOME> will be used in this document to refer to the root directory of SJSAS.

3. Read the Quick Start Guide at: <SJSAS_HOME>/docs/QuickStart.html on starting and stopping the server.

To install *JAX-WS 2.1 with WSIT* on the server:

1. Create an AS_HOME environment variable pointing to <SJSAS_HOME>.
2. Run in the <JAXWS_HOME> directory: `ant -f wsit-on-glassfish.xml install`

This command copies the *webservices-rt.jar* and *webservices-tools.jar* files from the <JAXWS_HOME>/lib directory to the <SJSAS_HOME>/lib directory.

It also copies the *webservices-api.jar* file from the <JAXWS_HOME>/lib directory to the <SJSAS_HOME>/lib/endorsed directory.

SJSAS 9 comes with JAX-WS 2.0, so we need to ensure the Java classes from the *JAX-WS 2.1 with WSIT* version are used instead of the older version. This is done by placing the *JAX-WS 2.1 with WSIT* implementation JAR files at the front of the server's classpath. The following steps will specify the necessary *JAX-WS 2.1 with WSIT* JAR files in the server's classpath prefix.

1. Start the application server if it is not running.
2. Log into the *Admin Console* for the application server.
3. Click on the *Application Server* link in the left-hand frame.
4. Click on the *JVM Settings* tab in the right-hand frame.
5. Click on the *Path Settings* tab.
6. Add the following 2 lines of text to the *Classpath Prefix* text area.


```

      ${com.sun.aas.installRoot}/lib/webservices-rt.jar
      ${com.sun.aas.installRoot}/lib/webservices-tools.jar
      
```
7. Click on the *Save* button. The application server has to be re-started for it to recognise the changes.

To turn off the capture stack trace feature of JAX-WS, do the steps described in the next section for the property: `com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace=false`.

B.6.1 Specify System properties

1. Start the application server if it is not running.
2. Log into the *Admin Console* for the application server.
3. Click on the *Application Server* link in the left-hand frame.
4. Click on the *JVM Settings* tab in the right-hand frame.
5. Click on the *JVM Options* tab.
6. For each system property, click on the *Add JVM Option* button and in the empty *Value* text field that appears, type in the `-D` option and then the property, e.g.:


```
-DpropertyName=propertyValue
```
7. Click on the *Save* button. The application server has to be re-started for it to recognise the changes.

Appendix C: Key management

The term, *key store*, has 2 meanings in Java depending on the context. The first meaning refers to files containing security tokens, such as certificates and private keys. It is used in relation to general security tools and APIs. The second meaning is more specific, referring to the file containing an entity's own certificate and private key. It is differentiated from a *trust store*, which refers to the file containing other entities' public certificates. It is used in relation to Web service and SSL (Secure Sockets Layer) APIs. This appendix section uses the first meaning.

C.1 Key store types

The standard Java distribution supports 3 key store types:

- PKCS #12
 - PKCS #12 belongs to the Public-Key Cryptography Standards (PKCS) group of specifications developed by RSA Laboratories. PKCS #12 is a standard format for storing and transferring identity information, such as certificates and private keys [PKCS1999].
 - The standard name for this key store type is *pkcs12*.
- Java Key Store (JKS)
 - JKS is a proprietary Java format for storing security tokens [JCA].
 - The standard name for this key store type is *jks*.
- Java Cryptography Extension Key Store (JCEKS)
 - JCEKS is another proprietary Java format, but it has stronger protection for private keys than JKS [JCA].
 - The standard name for this key store type is *jceks*.

Other key store types can be supported in Java using the extensible mechanisms of the Java Cryptography Architecture (JCA) [JCA].

C.2 Tools

The Java Development Kit (JDK) comes with a *keytool* command-line tool for managing keys and certificates [KEYTOOL]. This tool is found in the *\$JDK_HOME/bin* directory. It allows you to create key stores, import certificates into the key stores, list the keys in a key store, create self-signed certificates and more. Read the documentation for the *keytool* tool for the installed JDK. The functionality and command-line arguments for the tool can differ between JDK versions.

There are also open source tools, like openSSL [OPENSSL] and EJBCA [EJBCA], which allows you to create keys and certificates.

Appendix D: Debugging

This appendix contains some hints about debugging Web services.

An important first step to debugging Web service problems is to identify the failure point.

- *Was a SOAP request sent by the client?* If not, there could be problems with the client-side configuration. For example, the key store's password can be wrong.
- *Did the SOAP request reach the Web service implementation class?* If not, the server might have problems with the SOAP request. For example, the request can be encrypted with a certificate that the server does not have a private key for.
- *Was a SOAP response sent by the server?* If not, the server-side configuration might have errors. For example, the server cannot find the encryption key for the client.

Although the error messages will help indicate the problems, there are additional mechanisms to further identify the failure point.

D.1 Use a SOAP handler

The SOAP handler mechanism discussed in section 4.4 is one way to see the incoming and outgoing SOAP messages for the JAX-WS client and server. The code below provides an example of a SOAP handler that logs incoming and outgoing SOAP messages.

```
package com.example.common;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

public class LoggingHandler implements SOAPHandler<SOAPMessageContext> {

    private static final Logger LOG = Logger.getLogger(LoggingHandler.class
        .getName());

    public boolean handleMessage(SOAPMessageContext context) {
        try {
            if (isOutboundMessage(context)) {
                LOG.info("Outgoing message: ");
            }
            else {
                LOG.info("Incoming message: ");
            }
            LOG.info(getSOAPXML(context.getMessage()));
        }
        catch (Exception e) {
            // Handle exception
        }
        return true;
    }

    public boolean handleFault(SOAPMessageContext context) {
        try {
            if (isOutboundMessage(context)) {
                LOG.info("Outgoing fault: ");
            }
            else {
                LOG.info("Incoming fault: ");
            }
        }
    }
}
```

```

        LOG.info(getSOAPXML(context.getMessage()));
    }
    catch (Exception e) {
        // Handle exception
    }
    return true;
}

... // Empty implementations of methods for the SOAPHandler interface

private String getSOAPXML(SOAPMessage message) throws Exception {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    message.writeTo(out);
    out.flush();
    return new String(out.toByteArray());
}

private boolean isOutboundMessage(MessageContext context) {
    Boolean result = (Boolean) context
        .get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
    return result.booleanValue();
}
}

```

On the client-side SOAP handlers are declared in a binding file that is passed to the *wsimport* tool. This was described in section 4.2. The example below shows the declaration of the logging SOAP handler in a binding file.

```

<bindings xmlns="http://java.sun.com/xml/ns/jaxws"
  xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
  wsdlLocation="... ">
  <bindings node="wSDL:definitions">
    <jws:handler-chains xmlns:jws="http://java.sun.com/xml/ns/javaee">
      <jws:handler-chain>
        <!-- Other handlers in the handler chain -->
        <jws:handler>...</jws:handler>
        <jws:handler>
          <jws:handler-class>com.example.common.LoggingHandler</jws:handler-class>
        </jws:handler>
      </jws:handler-chain>
    </jws:handler-chains>
  </bindings>
</bindings>

```

On the server-side, SOAP handlers are declared in the *sun-jaxws.xml* configuration file, which was described in section 5.5.2.1. The example below shows the declaration of the logging SOAP handler in the *sun-jaxws.xml* file.

```

<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
  <endpoint ... >
    <jws:handler-chains xmlns:jws="http://java.sun.com/xml/ns/javaee">
      <jws:handler-chain>
        <jws:handler>
          <jws:handler-class>com.example.common.LoggingHandler</jws:handler-class>
        </jws:handler>
      </jws:handler-chain>
    </jws:handler-chains>
  </endpoint>
</endpoints>

```

D.2 Use an HTTP tool

A SOAP handler will only provide access to unsecured messages (e.g. without signatures and encryption). An HTTP debugging proxy tool or an HTTP monitor can be used to see the SOAP messages that are actually sent back and forth between client and server.

To send SOAP messages to an HTTP proxy tool, set the `http.proxyHost` and `http.proxyPort` system properties to the proxy's host and port when running the client application.

Note If the Web service URL uses `localhost` as the host name, JAX-WS does not route the SOAP messages through the specified proxy. To use a debugging proxy tool when the client and server are on the

same machine, use another way of referring to the machine, such as the IP address or the computer name in Windows, in the Web service URL.

D.3 View server logs

The error message in the SOAP fault that is returned to the client might not indicate the root cause of the error. Remember to check the server logs for more detailed information. The server log for SJSAS is at:

`$AS_HOME/domains/domain1/logs/server.log`