nehta

Example Technical Implementation of Interoperable Web Services with TLS

JAX-WS

Version 1.1 — 30 June 2010

National E-Health Transition Authority

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 © 2010, 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 ... 2
1.5.1 Terminology ... 2
1.6 Style Conventions ... 3
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 ... 10

3 Overview of Metro ... 14
3.1 General background ... 14
3.2 Technical overview ... 14
3.2.1 Client ... 14
3.2.2 Server ... 14
3.3 Requirements ... 15
3.3.1 Client deployment ... 15
3.3.2 Web service deployment ... 15
3.3.3 Client development ... 15
3.3.4 Web service development ... 15
3.4 Platform tested ... 15

4 Web service client ... 16
4.1 Generate the service interface classes from the WSDL files ... 16
4.2 Implement the client application ... 17
4.2.1 Load the configuration ... 17
4.2.2 Create a service object ... 18
4.2.3 Get a port object ... 19
4.2.4 Set request context properties on the port ... 19
4.2.5 Invoke Web service operations ... 20
4.3 Package the Web service Client ... 21
4.3.1 Directory structure ... 21
4.3.2 Create the JAR file ... 21
4.4 Run the Web service client ... 21
4.4.1 Set the classpath ... 21
4.4.2 Set the security key files ... 21

5 Web service ... 23
5.1 Generate the service interface classes from the WSDL files ... 23
5.2 Implement the Web service ... 23
5.3 Create the deployment descriptors ... 24
5.3.1 sun-jaxws.xml ... 24
5.3.2 web.xml ... 25

5.3.3 sun-web.xml 26
5.4 Package the Web service 26
5.4.1 Directory structure 26
5.4.2 Create the WAR file 26
5.5 Configure the server 27
5.5.1 Install Metro on the server 27
5.5.2 Configure TLS on the server 27
5.5.3 Set the security key files 28
5.6 Deploy the Web service 30
5.6.1 Deploy the WAR file 30
5.6.2 Accessing the Web service 31

Appendix A: References 32

Appendix B: Installation 34
B.1 Java Development Kit 34
B.2 Java Cryptography Extension Unlimited Strength Jurisdiction Policy Files 34
B.3 Ant 34
B.4 Metro 34
B.5 Servlet container 35
B.6 Glassfish 35
B.6.1 Specify System properties 36

Appendix C: Key management 37
C.1 Key store types 37
C.2 Tools 37

Appendix D: Debugging 39
D.1 Use a SOAP handler 39
D.2 View server logs 41

Document information

Change history

Version	Date	Comments
1.0	2009-04-01	Draft
1.1	2010-06-30	Use ATS 5820-2010 Web Services Profiles

This page is intentionally left blank.

1 Introduction

1.1 Background

The National E-Health Transition Authority (NEHTA) has recommended Web services as the mechanism for communication between organisations in Australia's e-health environment.

NEHTA has published a number of technical documents to support the use of Web services. These include the *Web Services Profiles* [ATS 5820—2010].

1.2 Purpose

This document is part of a series of documents that provide examples of how to build Web services and clients conforming to the *Web Services Profiles* [ATS 5820—2010]. The *Web Services Profiles* document [ATS 5820—2010] defines 3 profiles: "Web services base profile", "WS-Security profile" and "TLS security profile". This document explains a way of building Web services and clients that conform to the "Web services base profile" and "TLS security profile" using the Metro toolkit [METRO].

TLS (Transport Level Security) is a security protocol, also known as SSL (Secure Sockets Layer). It applies security to transport protocol packets. In contrast, WS-Security applies security at a higher layer by securing SOAP requests.

Metro is Sun's open source Web Services Stack that includes the reference implementation for the Java API for XML Web Services (JAX-WS) [JAXWS].

The main purpose of this document is to support the understanding and interpretation of the conformance criteria in the *Web Services Profiles* [ATS 5820—2010]. However, it can also assist programmers who are learning how to use Metro.

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 the Metro [METRO] toolkit and Web services secured with TLS [TLS1999].

There is an example technical implementation document for building Web services secured with TLS using the .NET Windows Communication Foundation (WCF) [WCF]. That example technical implementation can interoperate with this implementation, but it will not be discussed in this document.

Also available are example technical implementation documents for building Web services secured with WS-Security [WSS2006]. Since these example technical implementations conform to a different security profile, they will not interoperate with this implementation.

These example technical implementation documents 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 *Web Services Profiles* [ATS 5820—2010]. The criteria from [ATS 5820—2010] are referred to by their criterion number (e.g. "WS 3.1.1.1 (a)").

1.5 Definitions, acronyms, abbreviations

API	Application programming interface
CA	Certificate authority
DOM	Document Object Model
Glassfish	Open source JEE 5 application server from Sun. Previously distributed as the Sun Java System Application Server.
HTTP	Hypertext Transfer Protocol
Java SE	Java Standard Edition
Java EE	Java 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	Java SE Runtime Environment
Metro	Open source Web services stack from Sun
RPC	Remote Procedure Call
SJSAS	Sun Java System Application Server
SSL	Secure Sockets Layer
TLS	Transport Layer Security
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	A 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:

Italics	• Document titles • Program names, tool names • File names, directory paths • URLs • Keywords
`Monospace`	• XML fragments, namespaces, names of XML elements and types • Code fragments, names of classes, methods and fields • Assemblies, packages • Command-line calls and arguments • Configuration properties
`Monospace + Bold`	• Emphasis within XML and code fragments
"Double quotes"	• 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 Metro.

Chapter 4 describes how to create a Web service client using Metro.

Chapter 5 describes how to create a Web service using Metro.

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 Metro 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.

For the purpose of this example the business process for 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].

This is a request-response operation at the technical-level to comply with criterion WS 5.1.5.1 (a) from the *Web Services Profiles* [ATS 5820—2010].

[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.)

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 criterion WS 5.1.5.1 (a). 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.

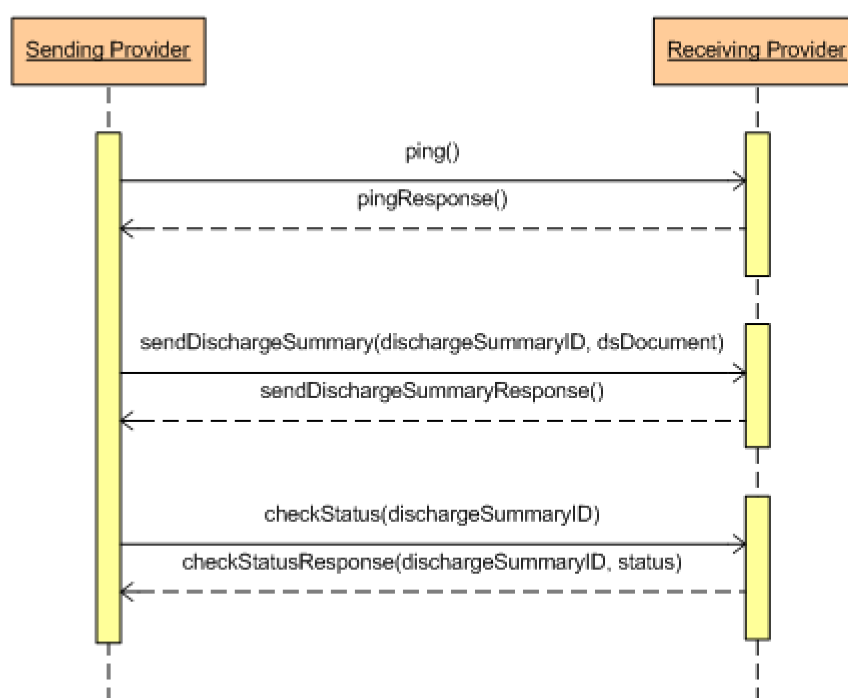

Figure 1: Example discharge summary business workflows

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].

The Web Service Definition Language (WSDL) [WSDL2001] and XML Schema Definition language (XSD) [XSD2004] are used to define the technical aspects of the service interface specification. Developers of Web services and clients do not have to create the WSDL and XSD files discussed in this section. These files will be created and published by the organisation producing the service interface specification. Developers may use these files as input to their toolkit when developing a Web service or client. Modifications may have to be made to the standard WSDL and XSD files to adapt to a particular toolkit. Modifications are permissible as long as the SOAP messages sent by the Web service or client conform to the WSDL and XSD specifications.

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://ns.nehta.gov.au/Example/WSP/DS/Xsd/DischargeSummary/1.0"
  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 to achieve the level of interoperability testing that it will be used for in these examples.

2.3.2.1 WSDL containing service interface information

Criterion WS 3.1.2.1 (a) 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.

```
<?xml version="1.0"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:tns="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
  targetNamespace="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
  name="DischargeSummaryReceiver">
```

It is a WSDL 1.1 document as required by criterion WS 3.1.1.1 (a). The `definitions` root element belongs to the WSDL 1.1 XML namespace, namely:

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

Following criterion WS 3.2.3.1 (a), the addressing information in the WSDL is described using *WS-Addressing 1.0 – Metadata* [WSAM2007], whose namespace is:

`http://www.w3.org/2007/05/addressing/metadata`

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

`http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0`

It is a URL and uses a slash character as the separator, as recommended by criteria WS 3.1.3.1 (b) and WS 3.1.3.1 (c) respectively. This service namespace is used as the target namespace of the WSDL, following criterion WS 3.1.3.1.4. 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.

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.

```
  <wsdl:types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
      xmlns:ds="http://ns.nehta.gov.au/Example/WSP/DS/Xsd/DischargeSummary/1.0"
      targetNamespace="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
      elementFormDefault="qualified">

      <xsd:import
        namespace="http://ns.nehta.gov.au/Example/WSP/DS/Xsd/DischargeSummary/1.0"
        schemaLocation="DischargeSummary.xsd"/>
```

The WSDL’s types section contains a schema defined using the W3C XML Schema language [XSD2004]. The target namespace of this schema is the service's namespace. Since the wrapper elements for the service’s operations are declared in this schema, the wrapper elements will belong to the service's namespace, as required by criterion WS 3.1.3.1 (h).

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

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

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

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

Since this ping operation takes no parameters and returns no results, both of these elements have an empty content model and no attributes.

To conform to criterion WS 5.1.4.1 (a), 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.

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

```
<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 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 criterion WS 5.1.5.1 (a).

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 criterion WS 5.1.4.1 (a). 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="invalidIdFaultMsg">
    <wsdl:part name="fault" element="tns:invalidIdFault"/>
  </wsdl:message>
```

2.3.2.1.4 Port Type

The portType section of the WSDL defines the three operations in the service. The operation definitions specify the structure of their input, output and fault messages by referencing the message definitions in the WSDL.

Following criterion WS 6.1.2.1 (a), the input, output and fault messages of all operations are assigned WS-Addressing `Action` values. The WS-Addressing `Action` attributes belong to the namespace of *WS-Addressing 1.0 - Metadata* [WSAM2007]. The values used for the WS-Addressing `Action` conform to the scheme set out in the following criteria: WS 3.1.3.1 (e), WS 3.1.3.1 (f) and WS 3.1.3.1 (g).

```
  <wsdl:portType name="DischargeSummaryReceiver">

    <wsdl:operation name="ping">
      <wsdl:input message="tns:pingInMsg"
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/pingRequest"/>
      <wsdl:output message="tns:pingOutMsg"
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/pingResponse"/>
    </wsdl:operation>

    <wsdl:operation name="sendDischargeSummary">
      <wsdl:input message="tns:sendDischargeSummaryInMsg"
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/sendDischargeSummaryRequest"/>
      <wsdl:output message="tns:sendDischargeSummaryOutMsg"
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/sendDischargeSummaryResponse"/>
      <wsdl:fault name="invalidIdFault" message="tns:invalidIdFaultMsg"
```

```
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/sendDischargeSummary/Fault/invalidIdFault"/>
    </wsdl:operation>

    <wsdl:operation name="checkStatus">
      <wsdl:input message="tns:checkStatusInMsg"
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/checkStatusRequest"/>
      <wsdl:output message="tns:checkStatusOutMsg"
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/checkStatusResponse"/>
      <wsdl:fault name="invalidIdFault" message="tns:invalidIdFaultMsg"
        wsam:Action="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0/
DischargeSummaryReceiver/checkStatus/Fault/InvalidIdFault"/>
    </wsdl:operation>

  </wsdl:portType>

</wsdl:definitions>
```

2.3.3 Non-functional attributes

The WSDL file can contain the non-functional attributes of a service in a formal specification. However, a WSDL file is not the complete documentation of a service. Some non-functional attributes of a service cannot be described formally within WSDL therefore requiring additional documentation.

2.3.3.1 WSDL containing service instance information

The non-functional attributes of a service are placed in a separate file as recommended by criterion WS 3.1.2.1 (a). This second WSDL file contains the service instance information. It specifies the concrete aspects of the service interface, such as how data is transported and how it is secured.

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

2.3.3.1.1 Header

In the second WSDL file, the start tag of the root element again contains all the XML namespaces that this document will use.

```
<?xml version="1.0"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:tns="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
  targetNamespace="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"
  name="DischargeSummaryReceiver">
```

Criterion WS 5.1.1.1 (a) 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/`

Following criterion WS 3.2.1.1 (a), the WSDL uses the *WS-Policy 1.5 Framework* [WSPL2007] to define the non-functional attributes of the service that can be specified formally within the WSDL file. The `wsp` prefix is used to refer to the namespace of *WS-Policy 1.5*, namely:

`http://www.w3.org/ns/ws-policy`

The security policies of the service are specified using *WS-SecurityPolicy 1.2* [WSSPL2007], which follows criterion WS 3.2.2.1 (a). The `sp` prefix is used to refer to the namespace of *WS-SecurityPolicy 1.2*, namely:

`http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702`

The addressing policies of the service are specified using *WS Addressing 1.0 - Metadata* [WSAM2007], which follows criterion WS 3.2.3.1 (a). The `wsam`

prefix is used to refer to the namespace of *WS-Addressing 1.0 - Metadata*, namely:

`http://www.w3.org/2007/05/addressing/metadata`

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

`http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0`

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

2.3.3.1.2 Addressing Policy

The addressing requirements of the service are declared using *WS-Addressing 1.0 - Metadata* [WSAM2007]. The addressing assertions are specified within a policy, defined using the *WS-Policy 1.5* [WSPL2007] policy language.

The service's addressing policies are specified in a policy called *AddressingPolicy*. The name *AddressingPolicy* is arbitrary - any unique name could have been used.

The `Addressing` assertion element indicates the use of WS-Addressing [WSAM2007], as per criterion WS 6.1.1.1 (a).

```
<wsp:Policy xml:id="AddressingPolicy">
  <wsam:Addressing/>
</wsp:Policy>
```

2.3.3.1.3 Security Policy

The security requirements of the service are specified within a policy, defined using the *WS-Policy 1.5* [WSPL2007] policy language. The *WS-SecurityPolicy 1.2* [WSSPL2007] is used to formally describe the security requirements.

The service's security policies are specified in a policy called *SecurityPolicy*.

```
<wsp:Policy xml:id="SecurityPolicy">
```

The `TransportBinding` assertion element declares the use of transport-level security.

```
  <sp:TransportBinding>
    <wsp:Policy>
```

The `TransportToken` assertion element within the `TransportBinding` assertion specifies the policy constraints for the security tokens used in transport-level security. The `HttpsToken` assertion specifies a requirement for security tokens that support the use of HTTPS. HTTPS refers to the use of the HTTP (Hypertext Transfer Protocol) over TLS (Transport Level Security) or SSL (Secure Sockets Layer). The `RequireClientCertificate` attribute specifies that the client must provide a certificate when establishing a TLS session. This matches with criteria WS 8.2.2.2 (a) and WS 8.2.2.2 (b), which requires the Web service to use a mutually authenticated connection.

```
      <sp:TransportToken>
        <wsp:Policy>
          <sp:HttpsToken RequireClientCertificate="true"/>
        </wsp:Policy>
      </sp:TransportToken>
    </wsp:Policy>
  </sp:TransportBinding>
</wsp:Policy>
```

2.3.3.1.4 Import

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

```
<wsdl:import location="DischargeSummaryReceiverInterface.wsdl"
  namespace="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"/>
```

2.3.3.1.5 Binding

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

The addressing and security policies that were defined in sections 2.3.3.1.2 and 2.3.3.1.3 have to be applied to the service. This is done using `wsp:PolicyReference` elements. The `wsp:PolicyReference` elements are applied to the WSDL `binding` element as recommended by criterion WS 3.2.1.1 (b).

Criterion WS 4.1.1.1 (a) recommends the use of HTTP 1.1 as the transport protocol. Although the particular HTTP version cannot be specified in the WSDL, the use of HTTP as the transport protocol can be specified by setting the `transport` attribute of the SOAP `binding` element to:

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

To comply with criterion WS 5.1.2.1 (a), 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`.

To comply with criterion WS 5.1.3.1 (a), `soapAction` values are not assigned to any operation. The `soapActionRequired` attributes are set to false to indicate that the service does not need `soapAction` values in the requests.

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

  <wsp:PolicyReference URI="#AddressingPolicy"/>
  <wsp:PolicyReference URI="#SecurityPolicy"/>

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

  <wsdl:operation name="Ping">
    <soap:operation style="document" soapActionRequired="false"/>
    <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" soapActionRequired="false"/>
    <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" soapActionRequired="false"/>
    <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>
```

2.3.3.1.6 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.example.com"/>
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

3 Overview of Metro

3.1 General background

Metro [METRO] is a Web services stack made up of 3 primary components, namely JAX-WS, JAXB and WSIT.

JAX-WS [JAXWS], the Java API for XML-Based Web Services, is a standard Java API for Web services, which was developed by the *Java Community Process JSR-224* [JSR224]. Metro contains an open-source reference implementation of the JAX-WS specification.

JAXB [JAXB], the Java Architecture for XML Binding, is a standard Java technology that maps Java objects to and from XML. It is a critical component of the implementation of Java-based Web services. It was developed by the *Java Community Process JSR-222* [JSR222]. A JAXB reference implementation is in Metro.

WSIT [WSIT], Sun's Web Services Interoperability Technologies, were developed to ensure interoperability between the Java platform and the *Windows Communication Foundation* (WCF) [WCF], from Microsoft.

3.2 Technical overview

3.2.1 Client

Using Metro, 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.

Metro 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 JAXB to marshal the Java method call and its parameters as a SOAP request. JAX-WS might also manipulate the SOAP request before it is sent, such as adding addressing headers. The XML SOAP request is placed in the body of an HTTP request. A secure HTTP connection is established with the Web service, and the HTTP request is sent over this secure connection.

The Web service will return an HTTP response that contains the XML SOAP response in its body. JAXB then unmarshals 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.

The deployment descriptor in the WAR file will specify whether a secure HTTP connection is required for communicating with the Web service. This is the case for Web services described in this document. If a client tries to make a Web service call over a non-secure HTTP connection, the Web container will reject the call returning an HTTP response with an HTTP error code.

When the Web container receives the request from the client over the secure HTTP connection, the request is passed to a servlet provided by the JAX-WS implementation. The SOAP request is retrieved from the body of the HTTP request. JAXB unmarshals 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, before it is passed to the servlet to return over the secure HTTP connection.

3.3 Requirements

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

3.3.1 Client deployment

- Metro Version 1.4
- Java Runtime Environment (JRE) 6
- Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files for JRE 6

3.3.2 Web service deployment

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

This document describes hosting a Web service in terms of the Glassfish Application Server because it is part of the platform used in testing the code and configuration fragments in the document. However, Glassfish is not required to host Metro Web services. Metro can be used with lightweight Java Web containers, such as Apache Tomcat, as well as other Java Enterprise Edition application servers.

3.3.3 Client development

- Metro 1.4
- Java Standard Edition Development Kit (JDK) 6

3.3.4 Web service development

The same software is required as the client development.

3.4 Platform tested

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

- Metro Version 1.4
- Glassfish Version 2.1
- JDK 6 Update 12
- JCE Unlimited Strength Jurisdiction Policy Files 6
- Ant 1.7.1
- Windows XP Professional SP3

4 Web service client

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

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.

The steps for building a Web service client are:

1. Generate the service interface classes from the WSDL files;
2. Implement the client application;
3. Package the client; and
4. Run the client.

4.1 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 Metro distribution. The code below assumes that there is a `METRO_HOME` environment variable that points to the root directory of the Metro toolkit.

```
<!-- Define path to Metro JAR files -->
<property environment="env"/>
<path id="metro.path">
  <fileset dir="${env.METRO_HOME}/lib" includes ="*.jar"/>
</path>

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

The following are commonly used 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;
- `fork`: set to true to fork the *wsimport* process into another Java virtual machine;
- `sourcedestdir`: location to place the generated source files; and
- `extension`: set to true to enable custom extensions.

  The Metro 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 *Web Services Profiles* [ATS 5820—2010].

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

4.2 Implement the client application

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

- Load the configuration;
- 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.FileReader;
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.RequestContextConstants;

public class ClientApp {

  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.out.println("Usage: java ClientApp [properties-file]");
    } else {
      // Load configuration properties
      String configFile = args[0];
      Properties configProperties = new Properties();
      configProperties.load(new FileReader(configFile));

      // Create service
      DischargeSummaryReceiverService service = createService();

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

      // Set request context properties on the port
      setRequestContextProperties(port, configProperties);

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

4.2.1 Load the configuration

In the simple client application in the example scenario, the configuration information comes from a properties file that was specified in a command-line argument. The contents of the properties file is read into a `java.util.Properties` object that will be used to configure the calls to the Web service in section 4.2.4.

```
public class ClientApp {

  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.out.println("Usage: java ClientApp [properties-file]");
    } else {
```

```
      // Load configuration properties
      String configFile = args[0];
      Properties configProperties = new Properties();
      configProperties.load(new FileReader(configFile));
      ...
    }
  }
  ...
}
```

The contents of an example configuration file for the client application is provided below.

```
service.url=https://localhost:8181/dsreceiver
```

The address of the Web service to call is specified in the `service.url` property. The protocol of the URL is *https*, which signifies that the HTTP connection is over SSL. The *localhost* host name assumes that the example Web service will be deployed in the same machine as the client. The 8181 port is the default port of the secure HTTP listener in a Glassfish application server.

4.2.2 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, which is done in `createService()` method in the code listing below.

```
public class ClientApp {

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

  private static DischargeSummaryReceiverService createService() {
    // Get annnotation from wsimport-generated DischargeSummaryReceiverService
    // class
    WebServiceClient annotation = DischargeSummaryReceiverService.class
        .getAnnotation(WebServiceClient.class);

    // Get details from the Web service's annotation
    String wsdlLocStr = annotation.wsdlLocation();
    String serviceName = annotation.name();
    String serviceNamespace = annotation.targetNamespace();

    // Search for the WSDL in the class path. Assume that the Web service's WSDL
    // files are in the JAR file containing the client app.
    URL wsdlLocUrl = getURLFromClasspath(wsdlLocStr);

    // Create the service
    QName serviceQName = new QName(serviceNamespace, serviceName);
    return new DischargeSummaryReceiverService(wsdlLocUrl, serviceQName);
  }
  ...
  private static URL getURLFromClasspath(String filePath) {
    URL result = null;
    if (filePath != null) {
      ClassLoader loader = Thread.currentThread().getContextClassLoader();
      result = loader.getResource(filePath);
    }
    return result;
  }
}
```

The generated `DischargeSummaryReceiverService` 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. In the example client, the WSDL location is retrieved from the classpath. Section 4.3 will show where to place the WSDL files when packaging the example client.

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.

4.2.3 Get a port object

The service class has a getter method for each port declared in the WSDL. These getter 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();
    ...
}
```

4.2.4 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.

```
public static void main(String[] args) throws Exception {
    ...
    // Set request context properties on the port
    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 host name verifier that permits 'localhost' in the Web service URL
  // (for development purposes)
  requestContext.put(
      "com.sun.xml.ws.transport.https.client.hostname.verifier",
      new PermitLocalhostVerifier());
}
```

The request context is retrieved by casting the port object to a `javax.xml.ws.BindingProvider` and calling the `getRequestContext()` method.

The WSDL containing the service instance information has a dummy value in the service's address, as discussed in section 2.3.3.1.6. This value can be overridden by setting the `BindingProvider.ENDPOINT_ADDRESS_PROPERTY` property in the request context. This is a built-in property of the JAX-WS API. This feature enables the client to dynamically select the service that it wishes to invoke. The client application in the example scenario sets the endpoint

address from the `service.url` property value in the properties file that was specified in a command-line argument.

When establishing a secure HTTP connection, Java will check the certificate provided by the server against the URL being accessed. The common name (CN) in the Subject's distinguished name of the server's certificate must match the host name in the Web service URL. To ease this restriction during software development, this behaviour can be overridden by providing a custom host name verifier.

Note: The host name verification is a critical security check. An example to override it is provided only for development purposes. It should not be overridden for production systems.

The custom host name verifier is provided by setting the `"com.sun.xml.ws.transport.https.client.hostname.verifier"` property in the request context. This property is specific to the Metro implementation; it is not part of the standard JAX-WS API.

An example custom host name verifier is provided below.

```
package com.example.ds.client;

import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;

public class PermitLocalhostVerifier implements HostnameVerifier {

  public boolean verify(String hostname, SSLSession session) {
    return hostname.equals("localhost");
  }

}
```

A host name verifier class must implement the `javax.net.ssl.HostnameVerifier` interface, which has one method called `verify`. Java will call the `verify` method of the host name verifier implementation when there is a mismatch between the host name in the URL and the common name in the server's certificate. The `verify` method will not be called if the host name and common name match.

This `verify` method will be passed the host name of the URL being accessed and the details of the SSL session being established (which includes the certificate provided by the server). It should return true to permit the secure HTTP connection, or false to reject the connection.

In this example host name verifier, it permits HTTP connections if *localhost* is used as the host name in the Web service's URL. This means that the Web service must be on the same machine as the client.

4.2.5 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
      String documentId = "...";
      ReceivedStatusType status = port.checkStatus(documentId);
  }
```

4.3 Package the Web service Client

4.3.1 Directory structure

The Web service client should be packaged in a JAR file.

The Java classes, including the stubs generated by the *wsimport* tool, should be placed in the root directory of the JAR file.

Since the code of the example client in section 4.2.2 looks for the WSDL file from the classpath, the WSDL and XSD files must be placed in the root directory of the JAR file.

4.3.2 Create the JAR file

The JAR file for the client can be created using the *jar* Ant task or the *jar* command-line tool. An example call to the *jar* Ant task is given below.

```
<jar destfile="dsreceiverClient.jar">
  <fileset dir="${classes.dir}">
    <include name="**/*.class" />
  </fileset>
  <fileset dir="${wsdl.dir}">
    <include name="*.wsdl" />
    <include name="*.xsd" />
  </fileset>
</jar>
```

4.4 Run the Web service client

4.4.1 Set the classpath

When running the client application, the client JAR file and the JAR files in *<METRO_HOME>/lib* directory should be in the classpath.

4.4.2 Set the security key files

4.4.2.1 Trust Store properties

In the TLS protocol, the client does not need to know the server's certificate before it communicates with the server. The server will send its certificate as part of the TLS handshake. However, the client must check that the server's certificate was issued by a trusted certificate authority before it can trust the identity asserted by the server's certificate.

The certificate authorities that the client trusts are placed in a key file called a *trust store*. *Appendix C:* provides instructions and notes on key files in Java.

The details of the trust store file must be specified in the following System properties:

- `javax.net.ssl.trustStore`: file location of the trust store;
- `javax.net.ssl.trustStoreType`: type of the trust store (*pkcs12*, *jks* or *jceks*); and
- `javax.net.ssl.trustStorePassword`: password for the trust store.

4.4.2.2 Key Store properties

To meet criterion WS 8.2.2.1 (a), the Web service client must support a mutually authenticated connection. To establish a mutually authenticated connection, the server will ask the client for its certificate.

The public and private keys for the client are placed in a key file called a *key store*. *Appendix C:* provides instructions and notes on key files in Java.

The details of the key store must be specified in the following System properties:

- `javax.net.ssl.keyStore`: file location of the key store;
- `javax.net.ssl.keyStoreType`: type of the key store (*pkcs12*, *jks* or *jceks*); and
- `javax.net.ssl.keyStorePassword`: password for the key store.

If there are multiple keys in the key store, Java will use the first key in the key store to establish the TLS connection. By default, it will assume that the password for the key matches that of the key store. If the passwords are not the same, a `java.security.UnrecoverableKeyException` will be thrown. A different password for the key can be specified in the following System property:

- `javax.net.ssl.keyPassword`: password for the key.

4.4.2.3 Setting System properties

When running the client application with the `java` command-line program, System properties are specified using the `-D` option, such as:

```
java … -Djavax.net.ssl.keyStore=… com.example.ds.client.ClientApp
```

When running the client application from an Ant script, System properties are specified using the `sysproperty` sub-element within the `java` task.

```
    <java classname="com.example.ds.client.ClientApp" fork="true">
      …

      <!-- TLS keys properties -->
      <sysproperty key="javax.net.ssl.keyStore"
                   value="${certificates.dir}/tls_client.jks" />
      <sysproperty key="javax.net.ssl.keyStoreType" value="JKS" />
      <sysproperty key="javax.net.ssl.keyStorePassword" value="password" />
      <sysproperty key="javax.net.ssl.trustStore"
                   value="${certificates.dir}/truststore.jks" />
      <sysproperty key="javax.net.ssl.trustStoreType" value="JKS" />
      <sysproperty key="javax.net.ssl.trustStorePassword" value="password" />
    </java>
```

5 Web service

This chapter describes how to build a Web service using Metro. 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. Generate the service interface classes from the WSDL files;
2. Implement the Web service;
3. Create the deployment descriptors;
4. Package the Web service;
5. Configure the server; and
6. 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.

Some steps are specific to the server software hosting the Web service. This example implementation document will explain these steps in terms of Glassfish, the open-source reference implementation for Java EE. Metro Web services can be hosted on any server with a servlet container. For those other server software, you will need to read their user guide to carry out some steps.

5.1 Generate the service interface classes from the WSDL files

The *wsimport* tool is also used to generate the service interface classes for the server. The information for this step in the client-side, explained in section 4.1, also applies to the server-side.

5.2 Implement the Web service

A Web service implementation class provides concrete method implementations for the operations in the WSDL.

An implementation of the Web service in the example scenario is given below.

```
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 InvalidIdFaultMsg{
    ...
  }

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

The Web service implementation class must have a `java.jws.WebService` Java annotation with an `endpointInterface` attribute. The value of this `endpointInterface` attribute must be the fully-qualified name of the port interface, which would have been generated from the WSDL in section 5.1. The `java.jws.WebMethod` Java annotation must be used to mark the methods that implement the operations in the WSDL.

The generated port interface contains Java method signatures that match the operations in the WSDL. Although it is not necessary, it is a good idea for the Web service implementation class to declare that it implements this generated port interface. By doing so, some discrepancies can be discovered at compile-time, such as missing method implementations.

5.3 Create the deployment descriptors

Deployment descriptors are configuration files for server-side engines and containers. A Web service needs to have a number of deployment descriptors.

5.3.1 sun-jaxws.xml

A *sun-jaxws.xml* file configures the Web service's endpoints in Metro. This file is specific to the JAX-WS reference implementation in Metro, and is not standardised. 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.ds.server.DischargeSummaryReceiverImpl"
    wsdl="WEB-INF/wsdl/DischargeSummaryReceiverService.wsdl"
    service="{http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0}Discharge
SummaryReceiverService"
    port="{http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0}Discharge
SummaryReceiver"
    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 5.2;
- *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 5.4. 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 *Web Services Profiles* [ATS 5820—2010]. The value for the SOAP 1.2 binding is: `http://java.sun.com/xml/ns/jaxws/2003/05/soap/bindings/HTTP/`

5.3.2 web.xml

The standard deployment descriptor for Java EE Web applications is a *web.xml* file. An example file for a Metro Web service is provided below.

```
<web-app>
  <!-- Declare Web service servlet context listener -->
  <listener>
    <listener-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>

  <!-- Declare Web service servlet -->
  <servlet>
    <servlet-name>DSRWebService</servlet-name>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Always map to Web service servlet -->
  <servlet-mapping>
    <servlet-name>DSRWebService</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>

  <!-- Declare security constraint for the Web service servlet -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>DSRWebService</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>

  <!-- Require client authentication with certificates -->
  <login-config>
    <auth-method>CLIENT-CERT</auth-method>
  </login-config>
</web-app>
```

Servlets are a Java EE technology that dynamically processes requests and builds responses. In traditional Web applications, servlets are used to dynamically generate HTML pages. Servlets are also used in Web services to process SOAP requests and construct SOAP responses.

A Metro Web service needs to declare some specific classes in its web deployment descriptor to use Metro correctly.

The `com.sun.xml.ws.transport.http.servlet.WSServletContextListener` class, which is part of the Metro implementation, must be declared as a servlet context listener in the Web application. A servlet context listener is notified when there are changes to the Web application's servlet context.

A servlet must be declared that is of type, `com.sun.xml.ws.transport.http.servlet.WSServlet`. This class is also part of the Metro implementation. Incoming HTTP requests for the Web application will be sent to this servlet, which will extract the SOAP request from the HTTP body, unmarshal it and call the appropriate method in the Web service implementation class from section 5.2. A SOAP response will be constructed from the result of the method call and returned in the body of the HTTP response.

The servlet declaration provides a name for the servlet to refer to it in the rest of the deployment descriptor. It also specifies that the servlet should be loaded when the Web service is deployed using the `load-on-startup` element.

A servlet mapping declaration determines which servlet handles which HTTP requests by specifying a mapping from a URL pattern to a particular servlet. The example *web.xml* file maps all requests to Metro's Web service servlet.

A security constraint declaration is needed to ensure the use of TLS with the Web service. The `web-resource-collection` element identifies what the security constraint applies to. It must refer to the Web service servlet. The `user-data-constraint` element defines what the security constraint is. A CONFIDENTIAL transport guarantee implies the requirement of TLS. The server will reject Web service requests that are not secured using TLS.

Since mutual authentication is required by criteria WS 8.2.2.2 (a) and WS 8.2.2.2 (b), the Web service must require that the client authenticates itself with a certificate. This is done in the web deployment descriptor by declaring a `login-conf` element, which is set to the CLIENT-CERT authentication method.

5.3.3 sun-web.xml

The Glassfish server uses a deployment descriptor called *sun-web.xml* for configurations that are specific to it. Since the example Web service is simple, it doesn't require additional Glassfish-specific configuration and this deployment descriptor is not needed. More complex Web services may require the additional configuration features available in a *sun-web.xml* file, such as setting resource references. See [SUNWEBXML] for the structure of the *sun-web.xml* file and [SUNWEBAPP] for information on how to configure Glassfish using the *sun-web.xml* file.

5.4 Package the Web service

5.4.1 Directory structure

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

```
/WEB-INF
     (Deployment descriptors, e.g. web.xml, sun-jaxws.xml, sun-web.xml)
     classes/
           (Java class files)
     lib/
           (JAR files)
     wsdl/
           (WSDL and XSD files)
```

Since there is no visible aspect to a Web service, all files are located within the *WEB-INF* directory. The deployment descriptors, which are discussed in 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. If Metro is installed on the server, the Metro JAR files do not need to be included.

There is no prescribed location for the WSDL and XSD files of a Web service. Metro 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.

5.4.2 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-*.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>
```

5.5 Configure the server

5.5.1 Install Metro on the server

Metro must be installed on the server that will host the Web service.

Appendix B: contains installation instructions and notes, including instructions on how to install Metro on a Glassfish server.

5.5.2 Configure TLS on the server

TLS has to be configured on the server software to conform to the "TLS security profile" in *Web Services Profiles* [ATS 5820—2010]. How this is done is specific to the server software. This section will only cover the Glassfish server.

HTTP listeners in Glassfish are listen sockets that wait for incoming HTTP requests on a specific port. There are 3 default HTTP listeners that come with a Glassfish installation:

- *http-listener-1*: is for HTTP traffic (transport security is turned off);
- *http-listener-2*: is for HTTPS traffic (transport security is turned on); and
- *admin-listener*: is for the *Admin Console* of the Glassfish server.

To view the HTTP listeners in the Glassfish installation:

1. Start the application server if it is not running.
2. Log in to the *Admin Console* for the application server.
3. In the tree pane on the left-hand side, select *Configuration*, then *HTTP Service* and then *HTTP Listeners*.

   In the right-hand panel, you should see a table that lists the HTTP listeners, whether they are enabled and which port they are listening on.

To configure TLS:

1. Select the secure HTTP listener, namely *http-listener-2* in a default installation.
2. Ensure that the *Security* checkbox is ticked.
3. Select the *SSL* tab.
4. Click on the *Client Authentication* checkbox so it becomes ticked.

   Mutual authentication is required by criteria WS 8.2.2.2 (a) and WS 8.2.2.2 (b). The server must require the client to authenticate itself.
5. Ensure that the *SSL2* checkbox is not ticked.

   SSL 2.0 must not be supported as per criterion WS 8.2.1.2 (b) because there are security flaws in the SSL 2.0 protocol.

   The *SSL3* checkbox should be left ticked. The use of the SSL 3.0 protocol is still permitted in the "TLS security profile".
6. Ensure that the *TLS* checkbox is ticked.

   TLS 1.0 must be supported as per criterion WS 8.2.1.2 (a).

7. Click on the *Add All >>* buttons for the *Common Ciphersuites* and *Ephemeral Diffie-Hellman Ciphersuites* lists. Ensure that no cipher suites are selected from the *40 bit and 56 bit Ciphersuites* list.

   The cipher suites in the *40 bit and 56 bit Ciphersuites* list either uses single DES or has a reduced key length to comply with previous US export restrictions. These cipher suites should be disabled to conform to the criteria WS 8.2.3.1 (c) and WS 8.2.3.1 (d).

8. Click on the *Save* button. The application server has to be re-started for it to recognise the changes.

5.5.3 Set the security key files

Setting the security key files for TLS is specific to the server software. Information in this section also only applies to the Glassfish server.

5.5.3.1 Trust Store properties

Like the client-side, there is a key file called a *trust store* that contains the certificate authorities that the server trusts. The details of the trust store are also specified in the following System properties:

- `javax.net.ssl.trustStore`: file location of the trust store;
- `javax.net.ssl.trustStoreType`: type of the trust store (*pkcs12*, *jks* or *jceks*); and
- `javax.net.ssl.trustStorePassword`: password for the trust store.

5.5.3.2 Key Store properties

Like the client-side, the public and private keys for the server are placed in a key file called a *key store*. The details of the key store are also specified in the following System properties:

- `javax.net.ssl.keyStore`: file location of the key store;
- `javax.net.ssl.keyStoreType`: type of the key store (*pkcs12*, *jks* or *jceks*); and
- `javax.net.ssl.keyStorePassword`: password for the key store; and
- `javax.net.ssl.keyPassword`: password for the key.

5.5.3.3 Security Keys in Glassfish

A default Glassfish installation will already have the TLS System properties set to default key store and trust store files.

The default trust store file is located at *<AS_HOME>/domains/domain1/config/cacerts.jks*, where <AS_HOME> is the directory of the Glassfish installation. The default password for this trust store file is *changeit*.

The default key store file is located at *<AS_HOME>/domains/domain1/config/keystore.jks*. The default password for this key store file is also *changeit*. The key for the server is the one in the key store with the *s1as* alias.

Note: Before making any changes, you should make copies of the *domain.xml*, *keystore.jks* and *cacerts.jks* files in the *<AS_HOME>/domains/domain1/config* directory. *domain.xml* is an XML file storing the Glassfish configuration for the server domain. Some errors resulting from changes made will cause the server to not start. It is highly recommended to make copies so that changes can be reverted.

5.5.3.3.1 *Change the master password*

The password for Glassfish key store and trust store files can be changed with the following steps:

1. Stop the application server if it is running.
2. In a command-line prompt, type:

   ```
   asadmin change-master-password --savemasterpassword=false
   ```

   You will be prompted for the new password for the key store and trust store files.

5.5.3.3.2 *Trust a new certificate authority*

Since mutual authentication is required, the server will ask the client for its certificate in the TLS handshake. The server will check that the client's certificate was issued by a trusted Certificate Authority (CA) before it trusts the identity asserted by the client's certificate.

Note: Trusting a Certificate Authority has important security implications. It should be treated with caution in production systems. The information below is primarily provided for development purposes when generated test certificates can be used.

To trust a new CA, the certificate of the CA must be added to the server's trust store by carrying out the following steps:

1. Stop the application server if it is running.
2. In a command-line prompt, change the directory to: *<AS_HOME>/domains/domain1/config*
3. Type:

   ```
   keytool –importcert
   -keystore cacerts.jks
   -storepass <master-password>
   –file <ca-cert-file>
   -alias <ca-cert-alias>
   ```

   where:

   master-password is the password for the trust store (it'll be the password set in section 5.5.3.3.1, or *changeit* if the master password wasn't changed);

   ca-cert-file is the certificate file for the trusted certificate authority; and

   ca-cert-alias is a short name to assign to this certificate in the trust store.

5.5.3.3.3 *Change the server's TLS key*

To change the server key, the new key must first be added to the server's key store.

1. Stop the application server if it is running.
2. In a command-line prompt, change the directory to: *<AS_HOME>/domains/domain1/config*
3. Type:

   ```
   keytool –importkeystore
   -destkeystore keystore.jks
   -deststorepass <master-password>
   -srckeystore <source-keystore-file>
   ```

```
-srcstoretype <source-keystore-type>
-srcstorepass <source-keystore-password>
-srcalias <source-key-alias>
-destalias <destination-key-alias>
-srckeypass <source-key-password>
-destkeypass <master-password>
```

where:

master-password is the password for the key store (it'll be the password set in section 5.5.3.3.1, or *changeit* if the master password wasn't changed);

source-keystore-file is the file containing the new private and public key pair;

source-keystore-type is the type of file containing the new key pair (e.g. *jks*, *pkcs12* or *jceks*);

source-keystore-password is the password for the new key pair;

source-key-alias is the short name associated with the key pair in the source key store;

destination-key-alias is the short name to associate the key pair in the server's key store; and

source-key-password is the password for the key in the source key store.

Note: The password for the key should match the password of the server's key store, namely the master password. Otherwise, the server may throw a `java.security.UnrecoverableKeyException`.

The secure HTTP listener in Glassfish must then be configured to use the new server key by carrying out the following steps:

1. Start the application server if it is not running.
2. Log in to the *Admin Console* for the application server.
3. In the tree pane on the left-hand side, select *Configuration*, then *HTTP Service* and then *HTTP Listeners*.
4. Select the secure HTTP listener, namely *http-listener-2* in a default installation.
5. Select the *SSL* tab.
6. In the *Certificate Nickname* field, type in the alias for the new server key in the key store (this would be the value of *destination-key-alias* from the previous set of steps).
7. Click on the *Save* button. The application server has to be re-started for it to recognise the changes.

5.6 Deploy the Web service

5.6.1 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 Glassfish, this directory is: *$AS_HOME/domains/domain1/autodeploy* for the default domain. Copy the WAR file created in section 5.4.2 to the auto-deploy directory.

5.6.2 Accessing the Web service

The Web service should be available at: https://<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 with a secure port 8181, the URL would be *https://XYZ:8181/dsreceiver*. The default port for the HTTPS protocol is 443. In the previous example, if the server XYZ had a secure HTTP port 443, the Web service's URL would be *https://XYZ/dsreceiver*.

Appendix A: References

[ATS 5820—2010] Standards Australia, *E-Health Web Services Profiles*.

[EJBCA] EJBCA, *EJBCA – The Java EE Certificate Authority*, http://ejbca.sourceforge.net.

[GIIWS2007] NEHTA, *Guidelines for Implementing Interoperable Web Services*, version 1.0, 28 March 2007.

[HANDLER] Sun Microsystems, *Metro 1.4 FCS – Handler*, https://metro.dev.java.net/nonav/1.4/docs/handlers.html.

[JCA] Sun Microsystems, *Java Cryptography Architecture (JCA)*, http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html.

[JAXB] Sun Microsystems, *Java Architecture for XML Binding (JAXB)*, https://jaxb.dev.java.net.

[JAXWS] Sun Microsystems, *Java API for XML Web Services (JAX-WS)*, https://jax-ws.dev.java.net.

[JSR224] JCP, *Java API for XML-Based Web Services*, 2.0, JSR-000224, http://jcp.org/aboutJava/communityprocess/pfd/jsr224/.

[JSR222] JCP, *Java API for XML Binding*, 2.0, JSR-000222, http://jcp.org/aboutJava/communityprocess/pfd/jsr222/.

[KEYTOOL] Sun Microsystems, *keytool - Key and Certificate Management Tool*, http://java.sun.com/javase/6/docs/technotes/tools/windows/keytool.html.

[METRO] Sun Microsystems, *Metro*, https://metro.dev.java.net.

[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.

[SUNWEBAPP] Sun Microsystems, *Sun Java System Application Server 9.1: sun-web-app*, http://docs.sun.com/app/docs/doc/819-3673/6n5sk1due?a=view#indexterm-467.

[SUNWEBXML] Sun Microsystems, *Sun Java System Application Server 9.1: The sun-web.xml File*, http://docs.sun.com/app/docs/doc/819-3673/6n5sk1d5a?a=view#beaql.

[TAIS2006] NEHTA, *Technical Architecture for Implementing Services: Concepts and Patterns*, version 1.0, 21 December 2006.

[TLS1999] IETF, The TLS Protocol, version 1.0, January 1999, http://www.ietf.org/rfc/rfc2246.txt

[WCF] Microsoft, *Windows Communication Foundation (WCF)*, http://netfx3.com/content/WCFHome.aspx.

[WSAM2007] W3C, *Web Services Addressing 1.0 – Metadata*, W3C Recommendation, 4 September 2007, http://www.w3.org/TR/2007/REC-ws-addr-metadata-20070904.

[WSDL2001] W3C, *Web Services Description Language (WSDL) 1.1*, W3C Note, 15 March 2001, http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[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.

[WSIT] Sun Microsystems, *Web Services Interoperability Technologies (WSIT)*, https://wsit.dev.java.net/.

[WSP2009] NEHTA, *Web Services Profile*, version 3.1 draft, 20 February 2009.

[WSPL2007] W3C, *Web Services Policy 1.5 – Framework*, W3C Recommendation, 4 September 2007, http://www.w3.org/TR/2007/REC-ws-policy-20070904.

[WSS2006] OASIS, *Web Services Security: SOAP Message Security 1.1*, OASIS Standard, 1 February 2006, http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[WSSP2006] NEHTA, *Web Services Standards Profile*, version 2.0, 20 November 2006.

[WSSPL2007] OASIS, *WS-SecurityPolicy 1.2*, OASIS Standard, 1 July 2007, http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-securitypolicy.pdf.

[XSD2004] W3C, *XML Schema Part 1: Structures*, W3C Recommendation, 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

Appendix B: Installation

B.1 Java Development Kit

1. Download and install JDK 6.

   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 *<JDK_HOME>*.

B.2 Java Cryptography Extension Unlimited Strength Jurisdiction Policy Files

The Java Cryptography Extension (JCE) provides cryptography services in the JDK. The JCE policy files in the JDK download are 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. Metro comes with Ant scripts to simplify its installation. To use these scripts, the Ant tool must be installed.

1. Download Ant 1.7.1.

   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 Metro

1. Download Metro 1.4.

   URL: https://metro.dev.java.net/1.4/

2. Unpack the downloaded JAR file with the command: `java -jar metro-installer.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 *metro* sub-directory within the directory containing the Metro installation JAR file.

3. Move the *metro* directory to the desired location.

   <METRO_HOME> will be used in this document to refer to this *metro* directory.

JDK 6 includes an older version of the JAX-WS reference implementation than the version of Metro 1.4. However, the API jars provided with JDK 6 are compatible with those in Metro 1.4.

B.5 Servlet container

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

1. Install the server software.
2. Install *Metro 1.4* on the server.

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

3. Turn off capture stack trace feature of Metro by setting the following System property:
   ```
   com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrac
   e = true
   ```

   If the capture stack trace feature is turned on, which it is by default, when a fault or exception occurs, Metro 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, Metro 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 Glassfish

This section explains how to do the steps described in the previous section in terms of Glassfish, Sun's Java Open Source EE reference implementation.

To install the server software:

1. Download Glassfish V2.1

   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.

   <AS_HOME> will be used in this document to refer to the root directory of Glassfish.

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

The following steps will install Metro on a Glassfish server:

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

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

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

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

B.6.1 Specify System properties

1. Start the application server if it is not running.
2. Log into the *Admin Console* of Glassfish.
3. In the left-hand tree pane, select *Configuration* and then *System Properties*.
4. Click on the *Add Property* button in the right-hand panel.
5. Specify the System property name in the *Instance Variable Name* column and the property value in the *Default Value* column.
6. Click on the *Save* button. The application server has to be re-started for it to recognise the changes.

Appendix C: Key management

Keys and certificates in Java are stored in files called key stores.

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. The second meaning is a differentiation in the role of the key files; the technical structure of the key files may be the same.

Most of this example implementation document has used the second meaning since most of the content was in relation to Web service and SSL APIs. However, this appendix section uses the first meaning since it covers general security tools.

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 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 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. Read the documentation for *keytool* for the installed JDK. The functionality and command-line arguments for the tool can differ between JDK versions.

Command-line instructions to carry out some common functionality with *keytool* in JDK 6 are listed below:

- List the keys and certificates in a key store

```
keytool -list
-keystore <file location of key store>
-storetype <type of key store>
-storepass <password for key store>
```

- Change the password for a key store

```
keytool -storepasswd
-keystore <file location of key store>
-storetype <type of key store>
-storepass <old password for key store>
-new <new password for key sore>
```

- Change the password for a key in a key store

```
keytool -keypasswd
-keystore <file location of key store>
-storetype <type of key store>
-storepass <password for key store>
-keypass <password for key>
-new <new password for key>
```

- Change the alias of a key or a certificate in a key store

```
keytool -changealias
-keystore <file location of key store>
-storetype <type of key store>
-storepass <password for key store>
-keypass <password for key>
-alias <old alias of key>
-destalias <new alias of key>
```

- Import a certificate into a key store

```
keytool -importcert
-keystore <file location of key store>
-storetype <type of key store>
-storepass <password for key store>
-file <file location of certificate>
-alias <alias to give certificate in key store>
```

- Import a key pair into a key store

```
keytool -importkeystore
-srckeystore <file location of source key store>
-srcstoretype <type of source key store>
-srcstorepass <password for source key store>
-destkeystore <file location of destination key store>
-deststoretype <type of destination key store>
-deststorepass <password for destination key store>
-srcalias <alias of key pair in source key store>
-destalias <alias of key pair in destination key store>
-srckeypass <password for key in source key store>
-destkeypass <password for key in destination key store>
```

There are also open source tools, like openSSL [OPENSSL] and EJBCA [EJBCA], which allow 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

SOAP handlers are a programmatic mechanism that provides access to SOAP messages before they are sent and after they are received [HANDLER]. The SOAP handler mechanism is one way to see the incoming and outgoing SOAP messages for Metro clients and services.

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 using the `binding` attribute. The example below shows a *wsimport* call that provides a binding file called *handlers.xml*.

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

The contents of the *handlers.xml* binding file are shown in the example below. It declares the use of the logging SOAP handler.

```
<bindings xmlns="http://java.sun.com/xml/ns/jaxws"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  wsdlLocation="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.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 5.3.1. 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 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 Glassfish's default domain is at: *<AS_HOME>/domains/domain1/logs/server.log*