

nehta

Example Technical Implementation of Interoperable Web services

WSE 3.0

Version 2.0 — 1 December 2008

Release

National E-Health Transition Authority Ltd

Level 25

56 Pitt Street

Sydney, NSW, 2000

Australia.

www.nehta.gov.au

Disclaimer

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

Document Control

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

Copyright © 2008, 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
1.1.1	Document History	1
1.2	Purpose.....	1
1.3	Scope	1
1.4	Intended audience	2
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	6
2.2.1	Discharge summary.....	6
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 WSE 3.0	16
3.1	General background	16
3.2	Technical overview.....	16
3.2.1	Extension points.....	16
3.3	Requirements.....	17
3.3.1	Client deployment	17
3.3.2	Web service deployment	17
3.3.3	Client development.....	17
3.3.4	Web service development	17
3.4	Platform used.....	18
4	Web service client	19
4.1	Modify the WSDL files.....	19
4.1.1	WSDL containing the service interface information	19
4.1.2	WSDL containing the service instance information.....	19
4.2	Generate the proxy from the WSDL files.....	20
4.3	Create the client application	21
4.4	Create the client custom policy assertion.....	21
4.4.1	Client output filter	22
4.4.2	Client input filter	25
4.5	Create the SOAP extension.....	26
4.5.1	AfterSerialize.....	27
4.5.2	BeforeDeserialize	28
4.5.3	AfterDeserialize.....	29
4.6	Implement the client application	30
4.6.1	Configure the client application	30
4.6.2	Create the proxy object.....	32
4.6.3	Invoke the Web service operations	33
5	Web service	34
5.1	Modify the WSDL files.....	34

5.2	Generate the service interface from the WSDL files.....	34
5.3	Create the Web service project	34
5.4	Create the custom service policy assertion	35
5.4.1	Create the service input filter.....	35
5.4.2	Service output filter.....	37
5.5	Create a custom exception class	41
5.5.1	Create the custom SOAP exception.....	41
5.5.2	Throw the exception	42
5.6	Implement the Web service.....	42
5.7	Configure the Web service	43
5.8	Package and deploy the Web service	45
5.8.1	Development server	45
5.8.2	Deploy with IIS.....	45
Appendix A: References.....		48
Appendix B: Installation.....		49
B.1	Visual Studio 2005.....	49
B.2	Web Services Enhancements 3.0 (WSE 3.0)	49
B.3	Windows HTTP Services Certificate Configuration Tool.....	49
B.4	Internet Information Services (IIS) (optional)	49
Appendix C: Key management.....		51
C.1	Setting up the certificate MMC	51
C.2	Installing certificates for the client application.....	51
C.2.1	Adding the client certificate	51
C.2.2	Adding the service certificate	52
C.2.3	Adding the CA	52
C.3	Setting up the service certificates	52
C.3.1	Adding the service certificate	52
C.4	Adding the CA.....	53
C.5	Pitfalls.....	53
Appendix D: Debugging.....		54
D.1	Diagnostics configuration	54
D.2	HTTP debugging proxy	54
D.3	Debugging a service that uses IIS.....	54

Document information

Change history

Version	Date	Comments
1.0	2007-07-03	Release
2.0	2008-12-01	Release

This page is intentionally left blank.

1 Introduction

1.1 Background

The National E-Health Transition Authority (NEHTA) has recommended Web services as the 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 Profile* [WSP2008].

1.1.1 Document History

Version 1.0 of this document was written to explain how to conform to the *Web Services Standards Profile* [WSSP2006] and *Guidelines for Implementing Interoperable Web Services* [GIIWS2007] using the Web Services Enhancements (WSE) 3.0 [WSE] from Microsoft.

The *Web Services Profile* [WSP2008] supersedes *Web Services Standards Profile* [WSSP2006] and *Guidelines for Implementing Interoperable Web Services* [GIIWS2007]. This document has been updated primarily to explain conformance to the *Web Services Profile* [WSP2008].

1.2 Purpose

This document provides an example of how Web services conforming to the *Web Services Profile* [WSP2008] can be implemented using a specific toolkit: Microsoft's Web Services Enhancements (WSE) 3.0 [WSE].

The End-to-end security profile in the *Web Services Profile* [WSP2008] lists conformance criteria for Web services and clients that are secured from end to end. This document explains a way of building Web services and clients that meet these criteria using the Microsoft Web Services Enhancements (WSE) 3.0 [WSE] toolkit.

The main purpose of this document is to support the understanding and interpretation of the conformance criteria in the *Web Services Profile* [WSP2008]. However, it can also assist programmers who are learning how to use WSE 3.0.

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

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

1.4 Intended audience

This document is intended for:

- Software developers; and
- System administrators.

It is expected that the reader is familiar with programming using C#, and 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 Profile* [WSP2008]. The criteria from [WSP2008] are referred to by their criterion number (e.g. "WS 3.1.1.1-1").

1.5 Definitions, acronyms, abbreviations

API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Secure HTTP
IDE	Integrated Development Environment
IIS	Internet Information Services
SDK	Software Development Kit
TCP	Transmission Control Protocol
WCF	Windows Communication Foundation
WSE	Web Service Enhancements
WSDL	Web Service Definition Language

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:

<i>Italics</i>	<ul style="list-style-type: none">• Document titles• Program names, tool names• File names, directory paths• URLs
Monospace	<ul style="list-style-type: none">• XML fragments, names of elements and types, namespaces• Code fragments, names of classes, methods, and fields• Assemblies, packages• Command-line calls and arguments• Configuration properties
Monospace + Bold	<ul style="list-style-type: none">• Emphasis for within XML and code fragments
"Double quotes"	<ul style="list-style-type: none">• Graphical user interface options

1.7 Overview

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

Chapter 3 provides a brief overview of WSE.

Chapter 4 describes how to create a Web services client program using WSE.

Chapter 5 describes how to create a Web service program using WSE.

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 WSE programs.

2 Example service

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

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

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

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

2.1 Organisational

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

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

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

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

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

2.1.1 Testing

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

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

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

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

This is a request-response operation at the technical-level to comply with criterion WS 5.1.5.1-1 from the *Web Services Profile* [WSP2008].

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 5.1.5.1-1. That is, a response is sent back, but it contains no business-level information.

2.1.3 Checking discharge summary status

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

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

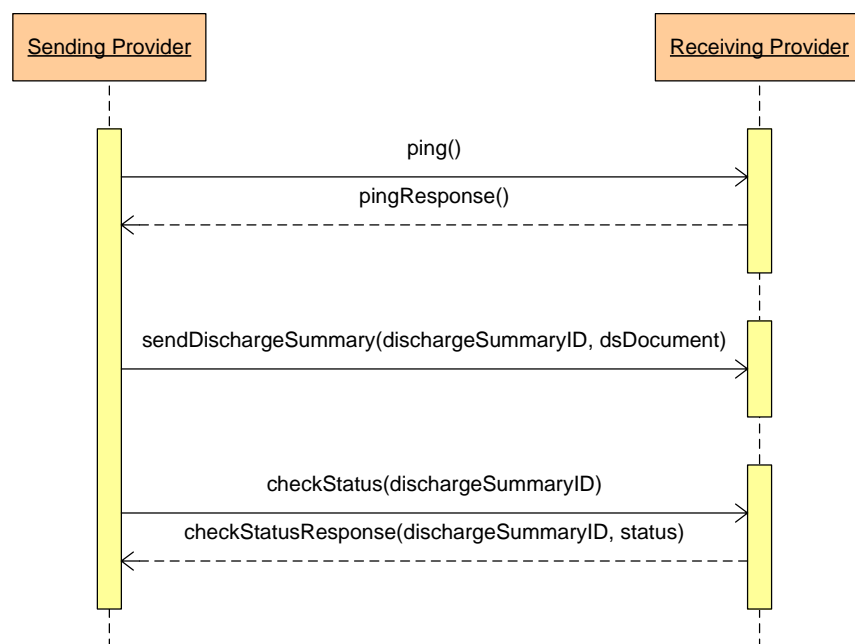


Figure 1: Example discharge summary business workflows

2.2 Informational

2.2.1 Discharge summary

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

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

2.2.2 Status

The possible status values for a discharge summary are:

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

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

2.3 Technical

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

2.3.1 Informational attributes

The XML Schema used to define a discharge summary document for the purpose of this example 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/Svc/Receiver/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 web 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 3.1.2.1-1 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.

It is a WSDL 1.1 document as required by criterion WS 3.1.1.1-1. The definitions root element belongs to the WSDL 1.1 XML namespace, namely:

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

Following criterion WS 3.2.3.1-1, 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 3.1.3.1-2 and WS 3.1.3.1-3 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.

```
<?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">
```

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. It contains a schema defined using the W3C XML Schema language [XSD2004].

The target namespace of the schema in the WSDL's `types` section 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 3.1.3.1-8.

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

```
<wsdl:types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://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 request and response elements for the ping operation are defined below. Since this operation takes no parameters and returns no results, both of these elements have an empty content model and no attributes.

To conform to criterion WS 5.1.4.1-1, this WSDL follows the wrapped convention. Thus, for all operations in this WSDL, the request element's name matches the operation's name, and the response element's name is the operation's name with a "Response" suffix.

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

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

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

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

Although the send discharge summary operation requires no business-level response, it has a response element, which has an empty content model and no attributes. This operation is modelled as a request-response operation at the technical level to satisfy criterion WS 5.1.5.1-1.

```
<xsd:element name="sendDischargeSummary">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="document" type="ds:DischargeSummaryType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

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

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

```
<xsd:element name="checkStatus">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="documentId" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="checkStatusResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="response" type="tns:ReceivedStatusType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

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

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

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

```
<xsd:element name="invalidIdFault">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="faultDescription" type="xsd:string"/>
      <xsd:element name="documentId" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
```

```

        </xsd: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 5.1.4.1-1. 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 7.1.2.1-1, 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 3.1.3.1-5, WS 3.1.3.1-6 and WS 3.1.3.1-7.

```

<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. A WSDL file is not the complete documentation of a service however and some non-functional attributes of a service cannot be described formally within WSDL and 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 3.1.2.1-1. 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.

Criterion WS 5.1.1.1-1 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 3.2.1.1-1, 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 3.2.2.1-1. 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 3.2.3.1-1. 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/Xsd/DischargeSummary/1.0
```

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

```
<?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">
```

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 7.1.1.1-1.

```
<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 `Wss11` assertion element declares the use of *WS-Security 1.1* as per criterion WS 6.2.1.1-1.

```
<sp:Wss11>
  <wsp:Policy>
```

The `MustSupportRefKeyIdentifier` element requires service providers and service clients to support key identifier references. This is needed because some certificates in the request and response messages have to be referenced using their subject key identifiers to conform to criteria: WS 6.2.7.1-2, WS 6.2.7.2-1 and WS 6.2.7.2-2.

```
  <sp:MustSupportRefKeyIdentifier/>
</wsp:Policy>
</sp:Wss11>
```

Criterion WS 6.2.3.1-1 requires that the SOAP body, the SOAP headers and the WS-Security timestamp be signed. The digital signature requirements of this criterion are met by providing the `SignedParts` assertion element in the policy.

The `SignedParts` element is used to indicate which parts of the SOAP messages are to be signed. When this element has no children, it specifies that the body and all headers targeted to the Ultimate Receiver role are to be signed [WSSPL2007].

An explicit assertion is not needed for signing the timestamp because if the timestamp is included, it implies that it must be signed [WSSPL2007].

```
<sp:SignedParts/>
```

Criterion WS 6.2.4.2-1 requires that the entire SOAP body is encrypted, while criterion WS 6.2.4.2-3 recommends that no SOAP headers are to be encrypted. These encryption requirements are met by providing the `EncryptedParts` assertion element in the policy.

The `EncryptedParts` element indicates which parts of a message should be encrypted. When this element has no children, it specifies that the body is to be encrypted. By default, SOAP headers are excluded from encryption.

```
<sp:EncryptedParts/>
```

The `AsymmetricBinding` element declares the use of public key cryptography as per criterion WS 6.1.1.1-1. It permits a child policy within itself to configure additional properties of the public key cryptography.

```
<sp:AsymmetricBinding>
  <wsp:Policy>
```

Criterion WS 6.2.2.2-1 requires that SOAP messages provide a Created timestamp in the WS-Security Timestamp element. The closest translation of this criterion is the `IncludeTimestamp` assertion, which requires the WS-Security Timestamp be provided in SOAP messages [WSSPL2007]. The `Created` element is optional within the WS-Security Timestamp element [WSS2006].

```
<sp:IncludeTimestamp/>
```

Criterion WS 6.2.3.1-2 stipulates that the digital signatures must be calculated over the entire element (i.e. including the start and end tags of the element). This behaviour is declared by the `OnlySignEntireHeadersAndBody` assertion element.

```
<sp:OnlySignEntireHeadersAndBody/>
```

The `EncryptSignature` assertion element declares that the digital signatures in messages must be encrypted, which matches criterion WS 6.2.4.2-2.

```
<sp:EncryptSignature/>
```

Criterion WS 6.2.5.2-2 requires the use of the Lax "Security Header Layout" property of `WS-SecurityPolicy` [WSSPL2007].

```
<sp:Layout>
  <wsp:Policy>
    <sp:Lax/>
  </wsp:Policy>
</sp:Layout>
```

An explicit assertion element is not required to declare the behaviour of signing before encrypting as per criterion WS 6.2.5.1-1. The default value of the "Protection Order" property of `WS-SecurityPolicy` [WSSPL2007] is "SignBeforeEncrypting".

The `Basic256Rsa15` assertion element declares the use of the `Basic256Rsa15` algorithm suite, as required by criterion WS 6.2.6.1-1.

```
<sp:AlgorithmSuite>
  <wsp:Policy>
    <sp:Basic256Rsa15/>
  </wsp:Policy>
</sp:AlgorithmSuite>
```

The `InitiatorToken` assertion element provides configuration properties for the service invoker's certificate.

In `WS-SecurityPolicy 1.2` [WSSPL2007], the initiator role belongs to the entity who sends the initial message and the recipient role belongs to the entity that is the target of the initial message. Using the terminology of this document, the service invoker is the initiator, and the service provider is the recipient.

Specifying an `AlwaysToRecipient` inclusion policy in the `IncludeToken` attribute on the `X509Token` element will meet criteria: WS 6.2.7.1-1 and WS-6.2.7.2-2. This token inclusion policy means that the service invoker's certificate is included in the messages to the service provider, which are namely the SOAP requests. In the other messages, namely the SOAP responses, the service invoker's certificate is not in the message itself, and is referenced indirectly.

The `WssX509V3Token10` element indicates that an X.509v3 token should be used for the service invoker's certificate, which follows criterion WS 6.1.2.1-1.

The `RequireKeyIdentifierReference` element specifies a key identifier reference should be used when the token is indirectly referenced. For an X.509v3 certificate, the key identifier reference is the subject key identifier. This assertion element is needed because criterion WS-6.2.7.2-2 requires that the subject key identifier be used to reference the service invoker's certificate in responses.

```
<sp:InitiatorToken>
  <wsp:Policy>
    <sp:X509Token sp:IncludeToken="http://docs.oasis-open.org/ws-sx/
ws-securitypolicy/200702/IncludeToken/AlwaysToRecipient">
      <wsp:Policy>
        <sp:RequireKeyIdentifierReference/>
        <sp:WssX509V3Token10/>
      </wsp:Policy>
    </sp:X509Token>
  </wsp:Policy>
</sp:InitiatorToken>
```

The `RecipientToken` assertion element is similar to the `InitiatorToken`, except that it applies to the security token of the service provider.

In the WSDL, the `RecipientToken` element has almost the same content as that of the `InitiatorToken` element, except that a *Never* inclusion policy is set on the `IncludeToken` attribute on the `X509Token` element. This token inclusion policy will meet criteria: WS 6.2.7.1-2 and WS 6.2.7.2-1. The service provider's certificate is never included in requests and responses, and is indirectly referenced.

```
<sp:RecipientToken>
  <wsp:Policy>
    <sp:X509Token sp:IncludeToken="http://docs.oasis-open.org/ws-sx/
ws-securitypolicy/200702/IncludeToken/Never">
      <wsp:Policy>
        <sp:RequireKeyIdentifierReference/>
        <sp:WssX509V3Token10/>
      </wsp:Policy>
    </sp:X509Token>
  </wsp:Policy>
</sp:RecipientToken>
</wsp:Policy>
</sp:AsymmetricBinding>
</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 3.2.1.1-2.

Criterion WS 4.1.1.1-1 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 5.1.2.1-1, 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 5.1.3.1-1, *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 WSE 3.0

3.1 General background

The Web Service Enhancements (WSE) 3.0 [WSE] is an add-on for Microsoft .NET and enables developers to create secure Web services based on open industry standards.

3.2 Technical overview

WSE 3.0 uses filters to intercept and modify SOAP messages when they're being sent and received. When a message is sent, the output filter adds the security headers and applies any transformations to the SOAP message (encryption) that are required. When a message is received, the input filter processes and removes the security headers and applies any transformations to the SOAP message (decryption) that are required.

WSE includes a number of turnkey security assertions. The turnkey security assertions support common scenarios, such as using mutual certificate and username/password authentication. WSE also allows custom policy assertions to be created when the turnkey assertions don't meet security requirements.

A client or Web service can be created from an existing WSDL (contract first) or by writing code. The .NET 2.0 framework provides a WSDL utility for creating .NET 2.0 proxies and service interfaces. WSE also provides a WSDL utility for creating proxies that can be secured.

Both clients and Web services can use WSE for securing messages. Web services that use WSE can be hosted within IIS or within an application using the WSE 3.0 messaging API.

3.2.1 Extension points

This section describes the extension points used in the example implementation.

3.2.1.1 SOAP extensions

SOAP extensions can be used to intercept and modify SOAP messages at different stages during message processing with .NET 2.0 clients and Web services.

There are four stages where a SOAP extension can intercept and modify a message:

1. Before serialisation: this stage allows manipulation of the object model of the outgoing message, before it is serialised into XML.
2. After serialisation: this stage allows manipulation of the XML of the outgoing message, before it is sent.
3. Before deserialisation: this stage allows manipulation of the XML of the incoming message, after it has been received, but before the XML is deserialised.
4. After deserialisation: this stage allows manipulation of the object model of the incoming message, after it has been deserialised from the XML.

Further information on SOAP extensions can be found in [MSDNSE].

3.2.1.2 Custom policy assertions

WSE provides a number of turnkey security assertions based on common scenarios. When the turnkey security assertions don't meet the security requirements, a custom policy assertion can be created.

A custom policy assertion can process security and modify incoming and outgoing messages.

Further information on custom policy assertions can be found in [MSDNPA].

3.3 Requirements

The following section details what software is required to build and deploy WSE 3.0 applications and services.

3.3.1 Client deployment

To run a WSE Web service client, the following software is needed:

- Microsoft Windows XP Home, Microsoft Windows XP Professional, Microsoft Windows 2000 Professional, Microsoft Windows 2000 Server, or Microsoft Windows Server 2003;
- Microsoft .NET Framework 2.0 Redistributable Package – Components required to run .NET 2.0 applications;
- Web Services Enhancements (WSE) 3.0 for Microsoft .NET Redistributable Runtime – Components required to run applications and services that use WSE 3.0.

3.3.2 Web service deployment

To run a WSE Web service, the client run-time requirements (section 3.3.1) plus the following are needed:

- Internet Information Services (IIS) 5.0, 5.1, or 6.0 – Set of Internet based services including a Web server that a Web service can be deployed on, allowing it to be accessible from a network. There are other ways to deploy a WSE 3.0 Web service that do not require IIS but these methods are not covered in this document.
- A tool to change the access privileges of the private keys and certificates in the Windows Certificate Store. For example, *Windows HTTP Server Certificate Configuration Tool (WinHTTPCertCfg)*.

3.3.3 Client development

To develop WSE Web service client programs, the client deployment requirements (section 3.3.1) plus the following are needed:

- Web Services Enhancements (WSE) 3.0 for Microsoft .NET – Toolkit required for the developing secure applications and services. This toolkit includes the development tools and the runtime components;
- A development environment for .NET programs. For example, Microsoft Visual Studio 2005, Microsoft Visual C# Express, or .NET SDK.

3.3.4 Web service development

The requirements for developing a Web service are the same as that for developing a Web service client (section 3.3.3).

3.4 Platform used

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

- Microsoft Windows XP Professional SP2;
- Visual Studio 2005;
- Web Service Enhancements (WSE) 3.0 for Microsoft .NET;
- Microsoft .NET Framework 2.0 Redistributable Package;
- Windows HTTP Server Certificate Configuration Tool; and
- IIS 5.0 (comes with Microsoft Windows XP Professional).

4 Web service client

This chapter describes how to build a Web service client using .NET 2.0 and WSE 3.0 that conforms to the *Web Services Profile* [WSP2008]. The aim of the Web service client is to invoke operations on a Web service.

The steps are:

1. Modify the WSDL files;
2. Generate the proxy from the WSDL files;
3. Create the custom client policy assertion;
4. Create the SOAP extension; and
5. Create a client program that implements the WSE proxies.

Each of these steps is described in more detail in the following sections.

4.1 Modify the WSDL files

WSE uses *WS-Addressing 2004* [WSA2004] to specify the addressing information in WSDL files. This W3C specification has been superseded by *WS-Addressing 1.0 - Metadata* [WSAM2007]. Since WSE implements the older specification, it only recognises the XML namespace of the older specification, namely <http://schemas.xmlsoap.org/ws/2004/08/addressing>.

4.1.1 WSDL containing the service interface information

4.1.1.1 WS-Addressing

As WSE does not support the WS-Addressing – Metadata [WSAM2007], the WS-Addressing *Action* attributes should be removed as this is not supported by WSE. A method is described later that allows the run-time service to continue to conform with [WSP2008].

```
<wsdl:definitions
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
  ...
  <!-- Removed as not supported by WSE
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  -->
  >
  ...
  <wsdl:portType name="DischargeSummaryReceiver">
    <wsdl:operation name="ping">
      <!-- Removed wsaw>Action attribute
      <wsdl:input message="tns:pingInMsg" wsaw>Action="http://..." />
      -->
      <wsdl:input message="tns:pingInMsg" />
    ...
```

4.1.2 WSDL containing the service instance information

4.1.2.1 WS-Policy

WSE does not support WS-Policy so all policy details should be removed from the WSDL containing the service instance information.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  ...
  <!--
  Removed the following namespace declarations:
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
```

```

-->
>
...
<!--
Removed the WS-Policy elements
<wsp:Policy wsu:Id="AddressingPolicy">
  <wsam:Addressing/>
</wsp:Policy>
<wsp:Policy wsu:Id="SecurityPolicy">
  ...
</wsp:Policy>
-->
...
</wsdl:definitions>

```

4.1.2.2 SOAP Actions

WSE populates the WS-Addressing Action values from the `soapAction` attributes in the WSDL. Therefore, the action values need to be added to the operations within the binding using the `soapAction` attribute. A method is described later that allows the run-time service to continue to conform with [WSP2008].

```

<wsdl:operation name="sendDischargeSummary">
  <!--
  Added soapAction Attribute
  <soap:operation style="document" />
  -->
  <soap:operation style="document" soapAction="http://..." />
  <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>

```

4.2 Generate the proxy from the WSDL files

The proxy is used to invoke operations on a remote Web service. The proxy consists of the interface providing the client operations plus the types and faults used by these operations. The proxy takes care of serialising the operation calls to SOAP requests that are sent to the Web service, and deserialising the SOAP responses received from the Web service.

To generate a WSE 3.0 proxy from the WSDL files, use the `wsewsdl3` command-line utility provided by WSE 3.0. The tool can be found in the `<WSE installation directory>\Tools` directory.

The following call to `wsewsdl3` generates the proxy code from the WSDL files:

```

wsewsdl3 DischargeSummaryReceiver.wsdl
  DischargeSummaryReceiverInterface.wsdl
  DischargeSummary.xsd
  /type:webClient
  /protocol:SOAP12
  /namespace Nehta.Example.DS.Client

```

The first three parameters specify the WSDL and XML Schema files for the Web service. It should be noted that all imported files need to be listed on the command line, not just the root WSDL file containing the service.

The other parameters are:

- `/type:` specifies the type of proxy to generate. This is set to `webClient` to generate a proxy that uses HTTP as the transport. This satisfies the *Web Services Profile* [WSP2008] criterion WS 4.1.1.1-1.
- `/protocol:` specifies what protocol to use. This is set to SOAP 1.2 to comply with criterion WS 5.1.1.1-1.

- `/namespace`: specifies what namespace the generated source file will be in. The generated class will be enclosed within the specified namespace. In this example, the C# namespace is set to `Nehta.Example.DS.Client`.

The `wsewsd/3` tool generates a single C# file that contains the proxy code. The proxy code contains all the classes needed to invoke operations on the Web service. The generated source file can be added to a client project without any modifications.

4.3 Create the client application

The client application can be any type of application. In this example it'll be a command-line application. This can be created from within Visual Studio.

4.4 Create the client custom policy assertion

The custom client policy assertion is used to process client requests and responses using custom input and output filters.

The client policy assertion is implemented by creating a class that extends the `SecurityPolicyAssertion` class and overriding the client input and output filter creation methods. The client output filter, discussed in section 4.4.1, processes requests sent by the client to the Web service. The client input filter, discussed in section 4.4.2, processes responses received by the client from a Web service.

In this example, the client policy assertion class contains two properties, `ClientToken` and `ServiceToken`. These are the security tokens that the client output filter will use to sign and encrypt the SOAP message.

The example client policy assertion class is shown below:

```
using System;
using System.Xml;

using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Design;
using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Security.Tokens;

namespace Nehta.Example.DS.Client.PolicyAssertion {
    public class ExampleDSClientPolicyAssertion : SecurityPolicyAssertion {
        private SecurityToken clientToken;
        private SecurityToken serviceToken;

        // Get and set the client token used for signing
        public SecurityToken ClientToken {
            get { return clientToken; }
            set { clientToken = value; }
        }

        // Get and set the service token used for encrypting
        public SecurityToken ServiceToken {
            get { return serviceToken; }
            set { serviceToken = value; }
        }

        // Filter creation methods

        public override SoapFilter CreateClientInputFilter(FilterCreationContext fcc)
        {
            return new ClientInputFilter(this, fcc);
        }

        public override SoapFilter CreateClientOutputFilter(FilterCreationContext fcc)
        {
            return new ClientOutputFilter(this, fcc);
        }

        public override SoapFilter CreateServiceInputFilter(FilterCreationContext fcc)
        { return null; }
    }
}
```

```

    public override SoapFilter CreateServiceOutputFilter(
        FilterCreationContext fcc)
    { return null; }
}

```

4.4.1 Client output filter

The client output filter processes the SOAP request messages that are sent from the client.

In this example, the output filter performs these tasks:

- Convert the WS-Addressing namespace from 2004 to 1.0;
- Specify the signing configuration; and
- Specify the encryption configuration.

The client output filter is implemented by creating a class that extends from the `SendSecurityFilter` class and overriding the `SecureMessage` method with the custom behaviour. The client and service tokens are passed from the custom client policy assertion and will be used to secure the SOAP request. The `SecureMessage` method is invoked by WSE before the request is sent to the Web service. The `SoapEnvelope` parameter provides access to the SOAP request before it has been secured by WS-Security. The `Security` parameter allows the security to be configured.

```

using System;
using System.Collections.Generic;
using System.Security.Cryptography.X509Certificates;
using System.Xml;
using System.Diagnostics;
using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Design;
using Microsoft.Web.Services3.Security.Tokens;
using Microsoft.Web.Services3;

namespace Nehta.Example.DS.Client.PolicyAssertion.Filter {

    class ClientOutputFilter : SendSecurityFilter {
        private SecurityToken clientToken;
        private SecurityToken serviceToken;

        public ClientOutputFilter(
            ExampleDSClientPolicyAssertion clientAssertion, FilterCreationContext fcc)
            : base(clientAssertion.ServiceActor, true) {
            clientToken = clientAssertion.ClientToken;
            serviceToken = clientAssertion.ServiceToken;
        }

        public override void
            SecureMessage(SoapEnvelope envelope, Security security) {

```

4.4.1.1 Add a unique identifier to the body.

A `wsu:ID` attribute is added to the body element of the SOAP message so it can be referenced for signing and encryption. The attribute is set to a GUID so it's unique within the message. The attribute is within the WS-Security Utility namespace: `"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"`.

```

// Generate a unique ID
string bodyElemId = Guid.NewGuid().ToString();

// Create the attribute
XmlAttribute bodyIdAttr = envelope.OwnerDocument.CreateAttribute(
    "Id",
    "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd");
bodyIdAttr.Value = bodyElemId;

```

```
// Add a 'wsu:Id' to the body
envelope.Body.Attributes.Append(bodyIdAttr);
```

4.4.1.2 Convert the WS-Addressing namespaces from 2004 to 1.0

WSE implements an older version of the WS-Addressing specification. When a request is being sent to a Web service, the WS-Addressing elements are automatically added by WSE to the SOAP message. These addressing elements are in the older namespace and must be converted to the newer namespace.

The `ConvertToWSAddressing10` method copies each addressing element in the old namespace to an element in the new namespace. The elements in the old namespace are then deleted.

```
// Create the WS-Addressing 1.0 headers and remove old headers
WSAddressingUtil.ConvertToWSAddressing10(envelope.DocumentElement);
```

The following is an example implementation of the `ConvertToWSAddressing10` method:

```
public static void ConvertToWSAddressing10(XmlNode soapMessage) {
    XmlNamespaceManager xnm =
        new XmlNamespaceManager(soapMessage.OwnerDocument.NameTable);
    xnm.AddNamespace("env",
        "http://www.w3.org/2003/05/soap-envelope");
    xnm.AddNamespace("wsa2004",
        "http://schemas.xmlsoap.org/ws/2004/08/addressing");

    // Get the SOAP header node
    XmlNode headerNode =
        GetXmlNode(soapMessage, "/env:Envelope/env:Header", xnm);

    // Find the old addressing nodes
    XmlNodeList oldAddressingNodes =
        headerNode.SelectNodes("/env:Envelope/env:Header/wsa2004:*", xnm);
    if (oldAddressingNodes.Count == 0) {
        throw new Exception("No 2004 addressing elements found");
    }
    else {
        // Convert the addressing nodes from the 2004 namespace
        List<XmlNode> existingNodes = new List<XmlNode>();
        foreach (XmlNode oldAddrNode in oldAddressingNodes) {
            existingNodes.Add(oldAddrNode);

            // Copy the node changing its namespace
            XmlNode cloneNode = CopyNodeToNamespace(oldAddrNode,
                "wsa10", "http://www.w3.org/2005/08/addressing",
                true, false);

            headerNode.AppendChild(cloneNode);
        }

        // Remove the 2004 addressing nodes
        foreach (XmlNode delNode in existingNodes) {
            headerNode.RemoveChild(delNode);
        }
    }
}
```

The `CopyNodeToNamespace` method copies an element into a new namespace including any attributes and child elements. Any text and whitespace is copied without modification.

4.4.1.3 Specify the signing configuration

The client output filter can specify the signing configuration for the request. This includes specifying which key is used to sign, which parts of the message are to be signed, and how the signing key is transported to the Web service.

A `MessageSignature` object is used to specify the details of how the message will be signed. The `MessageSignature` object is constructed using the client token which will in turn be used to generate the signature. As the `MessageSignature` results in a `Signature` element in the SOAP Header its `Id`

attribute is assigned a GUID to ensure it's uniquely referencable within the message. This attribute will be used to identify the `Signature` element during the encryption phase.

The signing certificate is included in the request message as shown by the first line of code below to satisfy criterion WS 6.2.7.1.1. The certificate can be used by a Web service to verify the signature and encrypt the response to the client.

```
// Add the client security token to the message
security.Tokens.Add(clientToken);

// Create the message signature object
MessageSignature msgSig = new MessageSignature(clientToken);
string signatureElemId = Guid.NewGuid().ToString();
msgSig.Signature.Id = signatureElemId;
// Set SignatureOptions to IncludeNone to allow custom signature handling.
msgSig.SignatureOptions = SignatureOptions.IncludeNone;
```

A `KeyInfo` object is created using the reference type to satisfy criterion WS 6.2.7.1-1. This uses a reference which points to the certificate that's included in the SOAP message.

```
// Create a KeyInfo so it is possible to specify the key reference type
KeyInfo keyInfo = new KeyInfo();
keyInfo.AddClause(new SecurityTokenReference(clientToken,
    SecurityTokenReference.SerializationOptions.Reference));
msgSig.KeyInfo = keyInfo;
```

A signature reference is added for the timestamp, body and all the converted WS-Addressing 1.0 elements.

```
// Create a signature reference for the timestamp element
SignatureReference sigRef = new SignatureReference();
sigRef.Uri = "#" + security.Timestamp.Id;
msgSig.AddReference(sigRef);

// Create a signature reference for the body element
sigRef = new SignatureReference();
sigRef.Uri = "#" + bodyElemId;
msgSig.AddReference(sigRef);

// Add IDs to the WS-Addressing 1.0 headers so they can be signed
List<string> addressElemIds = new List<string>();
XmlNodeList addressNodes =
    WSAddressingUtil.GetWSAddressing10Elements(envelope.DocumentElement);
foreach (XmlNode addressNode in addressNodes) {
    string addressId = Guid.NewGuid().ToString();

    // Create the attribute
    XmlAttribute addressAttr = envelope.OwnerDocument.CreateAttribute(
        "Id",
        "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd");
    addressAttr.Value = addressId;

    // Add a 'wsu:Id' to the address node
    addressNode.Attributes.Append(addressAttr);
    addressElemIds.Add(addressId);
}

// Add a signature reference for each addressing element
foreach (string id in addressElemIds) {
    sigRef = new SignatureReference();
    sigRef.Uri = HashChar + id;
    msgSig.AddReference(sigRef);
}

// Add the message signature
security.Elements.Add(msgSig);
```

This code only identifies the parts that are to be signed, but does not perform the actual signing operation. The actual signing will be performed by WSE after this method finishes.

4.4.1.4 Specify the encryption configuration

The client output filter is used to specify the details of how encryption will be applied to the request. This includes specifying which key is used to encrypt the message, the parts of the message to encrypt, and how the encryption key is obtained.

An `EncryptedData` object is used to specify the encryption details for the message and is constructed using the service token which will be used to perform the encryption. This service token is set by the client application that uses the client policy assertion.

```
// Create the encrypted data object
EncryptedData encData = new EncryptedData(serviceToken);

// Disable normal encryption of the body, so it can be controlled
encData.IncludeBodyElement = false;
```

The `ds:KeyInfo` object is created using the key identifier reference type to satisfy criterion WS 6.2.7.1-2. This uses the subject key identifier of the certificate.

```
// Create a KeyInfo so it is possible to specify the key reference type
KeyInfo encKeyInfo = new KeyInfo();
encKeyInfo.AddClause(new SecurityTokenReference(serverToken,
    SecurityTokenReference.SerializationOptions.KeyIdentifier));
encData.EncryptedKey.KeyInfo = encKeyInfo;
```

An encryption reference is added for the body and the signature to satisfy criteria WS 6.2.4.2-1 and WS 6.2.4.2-2.

```
// Create an encryption reference for the body
EncryptionReference encRef = new EncryptionReference("#" + bodyElemId);
encRef.Type = XmlEncryption.TypeURI.Content;
encData.AddReference(encRef);

// Create an encryption reference for the signature
encRef = new EncryptionReference("#" + signatureElemId);
encRef.Type = XmlEncryption.TypeURI.Element;
encData.AddReference(encRef);

// Add the encrypted data
security.Elements.Add(encData);
}
}
```

This code only identifies the parts that are to be encrypted, but does not perform the actual encryption. The actual encryption will be performed by WSE after this method finishes.

4.4.2 Client input filter

The client input filter processes the SOAP responses when they are received from a Web service.

In this example, the filter checks the message is signed and encrypted.

The client input filter is implemented by creating a class that extends from the `ReceiveSecurityFilter` class and overrides the `ValidateMessageSecurity` method with the custom behaviour. The `ValidateMessageSecurity` method is invoked by WSE when a response is received from a Web service. The `SoapEnvelope` parameter provides access to the SOAP message. The `Security` parameter allows the security configuration to be processed.

```
using System;

using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Design;
using Microsoft.Web.Services3;

namespace Nehta.Example.DS.ClientPolicyAssertion.Client.Filter {
```

```

class ClientInputFilter : ReceiveSecurityFilter {

    public ClientInputFilter(
        ExampleDSPolicyAssertion clientAssertion, FilterCreationContext fcc)
        : base(clientAssertion.ServiceActor, true)
    {
    }

    public override void
        ValidateMessageSecurity(SoapEnvelope envelope, Security security)
    {
    }
}

```

4.4.2.1 Check the message is signed and encrypted

The following checks if the request has been signed and encrypted.

```

if (security == null) {
    return;
}

bool isSigned = false;
bool isEncrypted = false;
foreach (ISecurityElement securityElem in security.Elements) {
    if (securityElem is MessageSignature) {
        if (isSigned) {
            throw new PolicyAssertionException("More than one signature found");
        }
        else {
            isSigned = true;
        }
    }
    else if (securityElem is EncryptedData) {
        if (isEncrypted) {
            throw new PolicyAssertionException("More than one encryption found");
        }
        else {
            isEncrypted = true;
        }
    }
}

if (!isSigned) {
    throw new PolicyAssertionException("SOAP message must be signed");
}

if (!isEncrypted) {
    throw new PolicyAssertionException("SOAP message must be encrypted");
}
}
}
}

```

Note that the above code does not verify the signature or perform the decryption. This is done by WSE after the `ValidateMessageSecurity` completes.

4.5 Create the SOAP extension

The SOAP extension is used to convert the WS-Addressing 1.0 elements of incoming messages to the draft WS-Addressing WSE implements.

The conversion is done in the SOAP extension and not in the client policy assertion because WSE processes the WS-Addressing headers before the client policy assertion is called.

The SOAP extension is implemented by creating a class that extends the `SoapExtension` class and overrides a number of methods with custom behaviour. The `ProcessMessage` method is invoked at different stages in the message processing.

```

namespace Nehta.Example.DS.Util.AddressSoapExtension {

    public class AddressSoapExtension : SoapExtension {
        public override object GetInitializer(LogicalMethodInfo methodInfo,
            SoapExtensionAttribute attribute)
        {
            return null;
        }

        public override object GetInitializer(Type WebServiceType) {
            return null;
        }

        public override void ProcessMessage(SoapMessage message) {
            switch (message.Stage) {
                case SoapMessageStage.BeforeSerialize:
                    break;

                case SoapMessageStage.AfterSerialize:
                    appStream.Position = 0;
                    Copy(_newStream, _oldStream);
                    break;

                case SoapMessageStage.BeforeDeserialize:
                    AddAddressingElements(message);
                    break;

                case SoapMessageStage.AfterDeserialize:
                    SetAddressHeadersDidUnderstand(message);
                    break;
            }
        }
    }
}

```

In this example, the SOAP extension does the following tasks:

- Convert the WS-Addressing from 1.0 to 2004; and
- Set the DidUnderstand flag on the 1.0 addressing elements.

4.5.1 AfterSerialize

WSE allows for chaining of multiple SOAP extensions, with each being executed in the order defined in configuration. Each extension is responsible for processing an inbound SOAP request, modifying the SOAP message and returning the modified stream after the `AfterSerialize` stage of processing.

It is typical to hold a reference to the original stream and create a new stream for the modified SOAP message. In the example below, this is done in the overridden `ChainStream` method. During the `AfterSerialize` stage of the overridden `ProcessMessage` method, the example `Copy` method is called which overwrites the contents of the original stream with the modified stream data. This in turn is chained to the next extension. When all extensions have been processed the resulting stream content is finally used as the SOAP message.

```

// Outgoing stream
private Stream _oldStream;
// Incoming stream
private Stream _newStream;

public override Stream ChainStream(Stream stream) {
    _oldStream = stream;
    _newStream = new MemoryStream();
    return _newStream;
}

void Copy(Stream from, Stream to) {
    TextReader reader = new StreamReader(from);
    TextWriter writer = new StreamWriter(to);
    writer.WriteLine(reader.ReadToEnd());
    writer.Flush();
}

```

4.5.2 BeforeDeserialize

4.5.2.1 Convert the WS-Addressing from 1.0 to 2004

WSE implements the older 2004 WS-Addressing specification [WSA2004]. When a response is received by the client that doesn't contain the 2004 addressing headers an exception is thrown. The 2004 addressing headers must be added by copying each of the WS-Addressing 1.0 elements into the WS-Addressing 2004 namespace.

The `AddWSAddressing2004Elements` method creates an XML document object from the inbound stream. This XML document is then processed by the `ConvertToWSAddressing2004` method. The `newStrm` parameter is then written to, with the results of the converted XML Document and new addressing elements.

Note: It is important to preserve the whitespace of the XML when loading the message into an XML document object. If whitespace is lost on elements that have been signed, such as the addressing elements, the signature will be invalidated.

```
private static void AddWSAddressing2004Elements(Stream oldStrm, Stream newStrm)
{
    // Read in the SOAP XML from the stream
    StreamReader readStr = new StreamReader(oldStrm);
    string soapMsg = readStr.ReadToEnd();

    // Load the stream into a DOM
    XmlDocument xmlDoc = new XmlDocument();
    xmlDoc.PreserveWhitespace = true;
    xmlDoc.LoadXml(soapMsg);

    // Check if the message contains a fault and the WSAddressing 2004
    // elements, if this is the case a low level fault has been thrown
    // from the service.
    if (WSAddressingUtil.IsFaultMessage(xmlDoc.DocumentElement) &&
        WSAddressingUtil.HasWSAddressing2004Elements(xmlDoc.DocumentElement)) {
        String errorMsg = null;
        String faultReason =
            WSAddressingUtil.GetFaultReason(xmlDoc.DocumentElement);
        if (faultReason == null) {
            errorMsg = "Invalid message was received";
        }
        else {
            errorMsg = faultReason;
        }

        throw new AddressException(faultReason);
    }
    else {
        // Add the addressing elements
        WSAddressingUtil.ConvertToWSAddressing2004(xmlDoc.DocumentElement);
    }

    // Write the document to the stream
    StreamWriter writeStr = new StreamWriter(newStrm);
    writeStr.Write(xmlDoc.InnerXml);
    writeStr.Flush();

    newStrm.Position = 0;
}
}
```

The `ConvertToWSAddressing2004` method copies the elements from the WS-Addressing 1.0 namespace to the draft 2004 WS-Addressing namespace.

The `CopyNodeToNamespace` method copies an element into a new namespace including any attributes and child elements. Any text and whitespace is copied without modification.

The existing WS-Addressing 1.0 elements cannot be deleted because they've been signed and this would invalidate the signature.

```
public static void ConvertToWSAddressing2004(XmlNode soapMessage,
```

```

bool deleteExisting)
{
    // Setup the namespace manager
    XmlNamespaceManager xnm =
        new XmlNamespaceManager(soapMessage.OwnerDocument.NameTable);
    xnm.AddNamespace("env",
        "http://www.w3.org/2003/05/soap-envelope");
    xnm.AddNamespace("wsa2004",
        "http://www.w3.org/2005/08/addressing");
    xnm.AddNamespace("wsa2004",
        "http://schemas.xmlsoap.org/ws/2004/08/addressing");

    // Find the header node
    XmlNode headerNode =
        GetXmlNode(soapMessage, "/env:Envelope/env:Header", xnm);

    // Get the existing 2004 addressing nodes
    XmlNodeList oldAddressingNodeList =
        headerNode.SelectNodes("/env:Envelope/env:Header/wsa2004:*", xnm);

    if (oldAddressingNodeList.Count == 0) {
        // Get the existing WS-Address 1.0 address nodes that will be copied
        XmlNodeList addressingNodeList =
            headerNode.SelectNodes("/env:Envelope/env:Header/wsa10:*", xnm);

        if (addressingNodeList.Count > 0) {
            // Copy the addressing elements from the 1.0 namespace to
            // the 2004 namespace
            List<XmlNode> existingNodes = new List<XmlNode>();
            foreach (XmlNode addrNode in addressingNodeList) {
                existingNodes.Add(addrNode);
                // Copy the node and change its namespace
                XmlNode clone = CopyNodeToNamespace(addrNode,
                    "wsa2004",
                    "http://schemas.xmlsoap.org/ws/2004/08/addressing",
                    true, false);

                headerNode.AppendChild(clone);
            }
        }
    }
}

```

Since some toolkits don't add the `To` WS-Addressing element within Web service responses, one has to be added. WSE requires the `To` element be present otherwise an exception is thrown. Its value is set to the 2004 WS-Addressing anonymous namespace.

```

// Check if a 'To' element exists and if there isn't add
// one using the anonymous namespace
XmlNodeList toNodeList =
    headerNode.SelectNodes("/env:Envelope/env:Header/wsa2004:To", xnm);

if (toNodeList.Count == 0) {
    // No 'To' element was found so add one
    XmlElement to = soapMessage.OwnerDocument.CreateElement(
        "wsa2004",
        "To",
        "http://schemas.xmlsoap.org/ws/2004/08/addressing");
    to.AppendChild(
        soapMessage.OwnerDocument.CreateTextNode(
            "http://schemas.xmlsoap.org/ws/dd2004/08/addressing/role/anonymous"
        ));

    headerNode.AppendChild(to);
}
}
}
}

```

4.5.3 AfterDeserialize

4.5.3.1 Set the `DidUnderstand` flag on the 1.0 addressing elements

The WS-Addressing 1.0 headers in the response have the `SOAP mustUnderstand` attribute set to `true`. This indicates to the receiver that the headers must be processed. This must be set to `false` so WSE does not try to process them otherwise an exception is thrown.

The following is an example implementation of the `SetAddressHeadersDidUnderstand` method:

```
// Set that each header was understood
private void SetAddressHeadersDidUnderstand(SoapMessage message) {
    // Set that each header was understood
    foreach (SoapHeader sh in message.Headers) {
        if (sh is SoapUnknownHeader) {
            SoapUnknownHeader suh = (SoapUnknownHeader) sh;

            // Check if the header is from the 1.0 addressing namespace
            if (suh.Element.NamespaceURI.Equals(
                Constants.WSAddressing10Namespace,
                StringComparison.OrdinalIgnoreCase)) {
                sh.DidUnderstand = true;
            }
        }
    }
}
```

4.6 Implement the client application

The steps for the client to invoke the Web service operations are:

1. Configure the client application;
2. Create the proxy object; and
3. Invoke the Web service operations.

4.6.1 Configure the client application

The following need to be configured:

1. WSE;
2. Revocation mode;
3. Encryption algorithms; and
4. SOAP extension.

The configuration is stored in the standard application configuration file called *app.config*.

4.6.1.1 WSE

WSE must be enabled before it can be used. The process below describes how to create a new configuration file to enable WSE.

1. Open the WSE configuration editor `WseConfigEditor3.exe` (the default location for this tool is `C:\Program Files\Microsoft WSE\v3.0\Tools\`).
2. On the General tab, check the box for "Enable this project for Web Services Enhancements".
3. Save the configuration file to *app.config*. This will overwrite the existing configuration file. It may be preferable to save the original configuration file to an alternate file name and then merge the two files.

The resulting configuration file should look like

```
<configuration>
  <configSections>
    <section name="microsoft.web.services3" type="..." />
  </configSections>
  <system.web>
    <webServices>
      <soapExtensionImporterTypes>
        <add
          type="Microsoft.Web.Services3.Description.WseExtensionImporter, ..." />
      </soapExtensionImporterTypes>
    </webServices>
  </system.web>
</configuration>
```

```

<compilation>
  <assemblies>
    <add assembly="Microsoft.Web.Services3, Version=3.0.0.0, ..." />
  </assemblies>
</compilation>
</system.web>
<microsoft.web.services3>
  <security>
    <x509 revocationMode="Online" />
  </security>
</microsoft.web.services3>
</configuration>

```

4.6.1.2 Revocation mode

The revocation mode determines if the certificates used in the security processing are checked for revocation. When doing testing, this should be disabled.

The following configuration change in the *app.config* file disables revocation checking:

```

<configuration>
  ...
  <microsoft.web.services3>
    <security>
      <x509 revocationMode="NoCheck" />
    </security>
  </microsoft.web.services3>
</configuration>

```

4.6.1.3 Encryption algorithms

The encryption algorithms need to be set to satisfy criterion WS 6.2.6.1-1. The key algorithm is set to RSA15 and the session key algorithm is set to AES256. The default signature and digest algorithms in WSE already satisfy criterion WS 6.2.6.1-1 and therefore do not need to be set specifically.

The following configuration change in the *app.config* file sets the encryption algorithms:

```

<configuration>
  ...
  <microsoft.web.services3>
    <security>
      <binarySecurityTokenManager>
        <add valueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3">
          <keyAlgorithm name="RSA15"/>
          <sessionKeyAlgorithm name="AES256"/>
        </add>
      </binarySecurityTokenManager>
      <x509 revocationMode="NoCheck" />
    </security>
  </microsoft.web.services3>
</configuration>

```

4.6.1.4 SOAP extension

The SOAP extension must be added to the application configuration so it can be used when the client sends and receives messages.

The following configuration change in the *app.config* adds the SOAP extension:

```

<configuration>
  ...
  <system.web>
    <webServices>
      <soapExtensionImporterTypes>
        <add type="Microsoft.Web.Services3.Description.WseExtensionImporter, ..." />
      </soapExtensionImporterTypes>
    </webServices>
  </system.web>
</configuration>

```

```

    <add
      type="Nehta.Example.DS.Util.AddressSoapExtension.AddressSoapExtension,
      Nehta.Example.Common" priority="1" group="0" />
    </soapExtensionTypes>
  </webServices>
</system.web>
...
</configuration>

```

Note that the full class name of the SOAP extension is used, including the assembly name.

4.6.2 Create the proxy object

1. Create the proxy object;
2. Set the Web proxy (optional);
3. Set the destination of the Web service; and
4. Set the client policy.

4.6.2.1 Create the proxy object

The first step in invoking operations on a Web service is to create an instance of the proxy class that was generated by the *wsewsdl/3* tool in section 4.1.

```

// Create an instance of the proxy
DischargeSummaryReceiverService serviceInstance =
  new DischargeSummaryReceiverService();

```

4.6.2.2 Set the Web proxy (optional)

When the client program is running behind a Web proxy, the address of the Web proxy needs to be set. This is done by creating a `WebProxy` object and setting it to the `Proxy` property of the proxy object.

The `WebProxy` class constructor has two parameters:

- The first parameter is `Address`, set this to the address and port of the proxy being used.
- The second parameter is the `BypassProxyOnLocal` parameter. If this is set to true, addresses that use `localhost` will bypass the proxy. When set to false, everything including `localhost` will go through the proxy.

```

serviceInstance.Proxy = new WebProxy("http://localhost:8888", false);

```

4.6.2.3 Set the destination of the Web service

The destination address of the Web service can be set on the proxy object. This is done by creating an `EndpointReference` object with the URL of the Web service and setting it to the `Destination` property of the proxy.

```

serviceInstance.Destination =
  new Microsoft.Web.Services3.Addressing.EndpointReference(
    new Uri("http://localhost/ExampleDSServer/Service.asmx"));

```

4.6.2.4 Set the client policy

The `SetPolicy` method on the proxy object is used to set the custom policy that'll be used by the client.

The client policy assertion is created and its two properties are set.

```

ExampleDSClientPolicyAssertion clientAssertion =
  new ExampleDSClientPolicyAssertion();

```

The security token that'll be used by the client output filter to sign the message must be set. The `CreateToken` method on the `X509TokenProvider` class can be used to retrieve tokens from the Windows certificate repository.

```
clientAssertion.ClientToken = X509TokenProvider.CreateToken(
    StoreLocation.CurrentUser, StoreName.My, "CN=MyClientTokenID");
```

The security token that'll be used by the client output filter to encrypt the message must be set. The `CreateToken` method on the `X509TokenProvider` class can be used to retrieve tokens from the Windows certificate repository.

```
clientAssertion.ServerToken = X509TokenProvider.CreateToken(
    StoreLocation.CurrentUser, StoreName.AddressBook, "CN=MyServiceTokenID");
```

The client `Policy` object is created and the client policy assertion is added.

```
Policy clientPolicy = new Policy();
clientPolicy.Assertions.Add(clientAssertion);
```

The client policy is set on the service instance object. This means that all requests and responses directed at the associated endpoint will use the policy.

```
// Set the policy to the proxy
serviceInstance.SetPolicy(clientPolicy);
```

4.6.3 Invoke the Web service operations

The client application invokes operations on the proxy object.

```
// Create and set parameters
checkStatus cs = new checkStatus();
cs.documentId = "someid"; // the unique ID of the discharge summary being queried

Debug.WriteLine("Calling checkStatus() document ID: " + );
try {
    checkStatusResponse csr = serviceInstance.checkStatus(cs);
    Debug.WriteLine("CheckStatus: id=" + cs.documentId + ", status=" + csr.response);
}
catch (...) {
    ""
}
```

4.6.3.1 Handling exceptions and faults

Within the WSDL, each operation specifies what faults can occur when calling that particular operation. When invoking a Web service method and a fault occurs, it can be caught using a standard try/catch block. Once caught, any normal error handling can take place.

5 Web service

This chapter describes how to build a Web service using .NET 2.0 and WSE 3.0 that conforms to the *Web Services Profile* [WSP2008]. The aim of a Web service is to provide a set of operations that a client can invoke.

The steps are:

1. Modify the WSDL files;
2. Generate the service interface from the WSDL files;
3. Create the Web service project;
4. Create the custom service policy assertion;
5. Create the custom exception class;
6. Implement the Web service;
7. Configure the Web service; and
8. Package and deploy the Web service.

5.1 Modify the WSDL files

The same modifications to the standard WSDL files will have to be made for the server-side as described in section 4.1

5.2 Generate the service interface from the WSDL files

The service interface defines the operations the Web service must implement. This includes the types and faults used in the operations. A Web service implements the service interface and exposes the operations to clients by providing an endpoint. A Web service client can then be used to invoke the operations. The service interface classes are generated from the WSDL files using the *wsdl.exe* utility that comes with the .NET 2.0 SDK.

The following call to *wsdl* generates the service interface from the WSDL files:

```
wsdl DischargeSummaryReceiver.wsdl
    DischargeSummaryReceiverInterface.wsdl DischargeSummary.xsd
    /protocol:SOAP12 /serviceInterface
```

The first three parameters specify the WSDL and XML Schema files for the Web service. It should be noted that all imported files need to be listed on the command line, not just the root WSDL file containing the service declaration.

The other parameters:

- */protocol*: indicates the protocol to use. SOAP 1.2 is used as the protocol in compliance with criterion WS 5.1.1.1-1.
- */serviceInterface*: indicates that a service interface should be generated and not a client.

This command will generate a single C# class file that contains the service interface. This generated interface will be used later for implementing the service itself which is described in section 5.6.

5.3 Create the Web service project

A .NET 2.0 Web service is created by creating an "ASP.NET Web Service" project from within Visual Studio. The project consists of the service implementation, *web.config* configuration file, and any referenced assemblies.

5.4 Create the custom service policy assertion

The custom service policy assertion is used to process requests and responses using custom input and output filters.

The service policy assertion is implemented by creating a class that extends the `SecurityPolicyAssertion` class and overrides the service input and output filter creation methods. The service input filter processes requests received by the Web service from a client. The service output filter processes responses sent by the Web service to the client.

```
using System;
using System.Xml;

using Microsoft.Web.Services3.Design;
using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Security.Tokens;

using Nehta.Example.DS.Server.PolicyAssertion.Filter;

namespace Nehta.Example.DS.Client.PolicyAssertion {
    public class ExampleDSClientPolicyAssertion : SecurityPolicyAssertion {
        private string serviceCertificateID;

        public string ServiceCertificateID {
            get { return serviceCertificateID; }
            set { serviceCertificateID = value; }
        }

        public override SoapFilter CreateClientOutputFilter(
            FilterCreationContext context)
        {
            return null;
        }

        public override SoapFilter CreateClientInputFilter(
            FilterCreationContext context)
        {
            return null;
        }

        public override SoapFilter CreateServiceInputFilter(
            FilterCreationContext context)
        {
            // Returns the custom service input filter
            return new ServiceInputFilter(this, context);
        }

        public override SoapFilter CreateServiceOutputFilter(
            FilterCreationContext context)
        {
            // Returns the custom service output filter
            return new ServiceOutputFilter(this, context);
        }
    }
}
```

The class also has a property `serviceCertificateID` for getting and setting the service certificate ID. The ID is used within the service output filter to retrieve the service certificate that is used to sign responses to clients. The ID is the subject distinguished name of the certificate.

In the following sections, the details of the service input and output filters will be described.

5.4.1 Create the service input filter

The service input filter processes the request messages when they are received by the Web service from the client.

In this example the filter performs two tasks:

- Check the request has been signed and encrypted; and

- Save information needed for the response.

The service input filter is implemented by creating a class that extends from the `ReceiveSecurityFilter` class and overrides the `ValidateMessageSecurity` method with the custom behaviour. The `ValidateMessageSecurity` method is invoked by WSE when a request is received from a client. The `SoapEnvelope` parameter provides access to the SOAP message. The `Security` parameter allows the security configuration to be processed.

```
using System;
using System.Web.Services.Protocols;

using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Design;
using Microsoft.Web.Services3.Security.Tokens;
using Microsoft.Web.Services3;

using Nehta.Example.DS.Util;

namespace Nehta.Example.DS.Server.PolicyAssertion {

    class ServerInputFilter : ReceiveSecurityFilter {

        public ServerInputFilter(ExampleDSPolicyAssertion exampleAssertion,
            FilterCreationContext fcc)
            : base(exampleAssertion.ServiceActor, false)
        {
        }

        public override void
            ValidateMessageSecurity(SoapEnvelope envelope, Security security)
        {
```

5.4.1.1 Check the request has been signed and encrypted

The following checks if the request has been signed and encrypted. It also stores the security token that was used to sign the request. This is used later to encrypt the response.

```
SecurityToken clientToken = null;
SecurityToken serviceToken = null;

foreach (ISecurityElement securityElem in security.Elements) {
    if (securityElem is MessageSignature) {
        if (clientToken != null) {
            throw new PolicyAssertionException("More than one signature found");
        }
        else {
            MessageSignature msgSig = securityElem as MessageSignature;
            clientToken = msgSig.SigningToken;
        }
    }
    else if (securityElem is EncryptedData) {
        if (serviceToken != null) {
            throw new PolicyAssertionException("More than one encryption found");
        }
        else {
            EncryptedData encData = securityElem as EncryptedData;
            serviceToken = encData.SecurityToken;
        }
    }
}

// Check the message contains a signature
if (clientToken == null) {
    throw new PolicyAssertionException("Incoming request was not signed");
}

// Check the message has been encrypted
if (serviceToken == null) {
    throw new PolicyAssertionException("Incoming request not encrypted");
}
```

Note that the above code does not verify the signature or perform the decryption. This is done by WSE after the `ValidateMessageSecurity` completes.

5.4.1.2 Save information needed for the response

When the Web service receives a secured request from a client, it needs to save the certificate that was used to sign the request so it can be used to encrypt the response. The `WS-Addressing Action` element value of the request also needs to be saved.

To pass information from the service input filter to the service output filter, the `OperationState` is used. A holder object can be saved into the `OperationState` from the service input filter and later retrieved from the `OperationState` in the service output filter. The `OperationState` can be accessed from the `SoapEnvelope` parameter passed to the `ValidateMessageSecurity` method.

The holder object is created and the action value and client token properties are set. This object is then set into the `OperationState`:

```
RequestState requestState =
    new RequestState(clientToken, envelope.Context.Addressing.Action.Value);
envelope.Context.OperationState.Set(requestState);
}
}
```

The following is an example of the `RequestState` holder class:

```
using System;

using Microsoft.Web.Services3.Security.Tokens;

namespace Nehta.Example.DS.Server.PolicyAssertion {
    class RequestState {
        private SecurityToken clientToken;
        private string actionValue;

        public RequestState(SecurityToken clientToken, string actionVal) {
            this.clientToken = clientToken;
            this.actionValue = actionVal;
        }

        public SecurityToken ClientToken {
            get { return this.clientToken; }
            set { this.clientToken = value; }
        }

        public string ActionValue {
            get { return this.actionValue; }
            set { this.actionValue = value; }
        }
    }
}
```

5.4.2 Service output filter

The service output filter processes the response messages before they are sent to the client.

In this example, the output filter performs these tasks:

- Modify the `WS-Addressing Action` element value;
- Convert the `WS-Addressing` from 2004 to 1.0;
- Specify the signing configuration; and
- Specify the encryption configuration.

The service output filter is implemented by creating a class that inherits from the `SendSecurityFilter` class and overrides the `SecureMessage` method with

the custom behaviour. The `SecureMessage` method is invoked by WSE before the response is sent to the client. The `SoapEnvelope` parameter provides access to the SOAP message before WS-Security is applied to it. The `Security` parameter allows the security SOAP header to be configured.

```
namespace ExampleDS.PolicyAssertion.Service {
    class ServiceOutputFilter : SendSecurityFilter {
        private string serviceCertificateID;

        public ServiceOutputFilter(ExampleDSPolicyAssertion serviceAssertion,
            FilterCreationContext fcc)
            : base(serviceAssertion.ServiceActor, false)
        {
            this.serviceCertificateID = serviceAssertion.ServiceCertificateID;
        }

        public override void
            SecureMessage(SoapEnvelope envelope, Security security)
        {
```

The holder object that was saved into the `OperationState` within the service input filter is retrieved so the action value and client token can be used.

```
RequestState requestState = envelope.Context.
    OperationState.Get<RequestState>();
if (requestState == null) {
    return;
}

SecurityToken clientToken = requestState.ClientToken;
string actionValue = requestState.ActionValue;
```

The service token used to sign the response is retrieved from the Windows certificate store using the `serviceCertificateID` property. This will be used later when specifying the signing configuration.

```
SecurityToken serviceToken =
    SecurityUtil.GetServiceToken(serviceCertificateID);
```

A `wsu:ID` attribute is added to the body element of the SOAP message so it can be referenced for signing and encrypting. The attribute is set to a GUID so it's unique within the message. The attribute is within the WS-Security Utility namespace: "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd".

```
// Create the unique ID
string bodyElemId = Guid.NewGuid().ToString();
// Add a 'wsu:Id' to the body
AddIdAttribute(envelope.Body, bodyElemId);
```

5.4.2.1 Modify the WS-Addressing Action element value

When WSE sends a response from a Web service to a client, it does not use the output WS-Addressing Action value specified in the WSDL. Instead, it appends the word "Response" to the value of the request Action. For example, if the request action is "http://example.com/test/pingRequest", the response action will be "http://example.com/test/pingRequestResponse", when it should be "http://example.com/test/pingResponse". This value must be modified to comply with criterion WS 3.1.3.1-6. The action value of faults is also handled incorrectly by WSE and needs to be modified to comply with criterion WS 3.1.3.1-7.

The `UpdateActionElement` method is used to modify the action value. The action value passed in is the one in the holder object that was retrieved from the `OperationState`. This has been set to the action value of the request.

```
// Update the action value
WSAddressingUtil.UpdateActionElement(envelope.DocumentElement, actionValue);
```

The following is an example implementation of the `UpdateActionElement` method:

```
public static void UpdateActionElement(XmlNode soapMessage,
    string actionValue)
{
    XmlNamespaceManager xnm =
        new XmlNamespaceManager(soapMessage.OwnerDocument.NameTable);
    xnm.AddNamespace(SoapEnvelopePrefix,
        Constants.SoapEnvelopeNamespace);
    xnm.AddNamespace(WSAddressing2004Prefix,
        Constants.WSAddressing2004Namespace);

    // Remove the "Request" text from the action
    actionValue = actionValue.Remove(actionValue.LastIndexOf(RequestText));

    if (!IsFaultMessage(soapMessage)) {
        actionValue = actionValue + ResponseText;
    }
    else {
        XmlNode faultNode =
            GetXmlNode(soapMessage, FaultDetailXPath, xnm);
        string faultName = faultNode.LocalName;
        actionValue = actionValue + ":" + FaultText + ":" + faultName;
    }

    XmlNodeList actionNodeList = soapMessage.SelectNodes(
        ActionElementXPath, xnm);

    if (actionNodeList.Count == 1) {
        XmlNode actionNode = actionNodeList[0];
        actionNode.InnerText = actionValue;
    }
}
```

5.4.2.2 Convert the WS-Addressing namespaces from 2004 to 1.0

WSE implements an older version of the WS-Addressing specification [WSA2004]. When a response is being sent to a client, the WS-Addressing elements are automatically added by WSE to the SOAP message. These addressing elements are in the older WS-Addressing namespace and must be converted to the newer namespace.

The `ConvertToWSAddressing10` method copies each addressing element in the old namespace to an element in the new namespace. The elements in the old namespace are then deleted. The details of this method are described in section 4.4.1.2.

```
// Add the addressing 1.0 elements
WSAddressingUtil.ConvertToWSAddressing10(envelope.DocumentElement, false);
```

5.4.2.3 Specify the signing configuration

The service output filter specifies the signing configuration for the response. This includes specifying which key is used to sign, which parts of the message are to be signed, and how the certificate associated with the signing key is identified or provided to the client.

A `MessageSignature` object is used to specify the signing configuration. The `MessageSignature` object is created using the service token. This is the token that'll be used to create the signature. The `Id` of the `MessageSignature` object is set to a GUID so the signature can be encrypted. The attribute is set to a GUID so it's unique within the message.

```
// Create the message signature object
MessageSignature msgSig = new MessageSignature(serviceToken);
string signatureElemId = Guid.NewGuid().ToString();
msgSig.Signature.Id = signatureElemId;
msgSig.SignatureOptions = SignatureOptions.IncludeNone;
```

A `KeyInfo` object is created using the key identifier type to satisfy criterion WS 6.2.7.2-2. This uses the subject key identifier of the certificate.

```
// Create a KeyInfo so it is possible to specify the key reference type
KeyInfo keyInfo = new KeyInfo();
keyInfo.AddClause(new SecurityTokenReference(clientToken,
    SecurityTokenReference.SerializationOptions.KeyIdentifier));
msgSig.KeyInfo = keyInfo;
```

A signature reference is added for the timestamp, body and all the converted WS-Addressing headers to satisfy criterion WS 6.2.3.1-1.

```
// Create a signature reference for the timestamp element
SignatureReference sigRef = new SignatureReference();
sigRef.Uri = "#" + security.Timestamp.Id;
msgSig.AddReference(sigRef);

// Create a signature reference for the body element
sigRef = new SignatureReference();
sigRef.Uri = "#" + bodyElemId;
msgSig.AddReference(sigRef);

// Create an ID for each address element add it to the element
List<string> addressIds = new List<string>();
XmlNodeList addressNodes =
    WSAddressingUtil.GetWSAddressing10Elements(envelope.DocumentElement);
foreach (XmlNode addressNode in addressNodes) {
    // Generate a GUID
    string addressId = Guid.NewGuid().ToString();
    // Add the attribute with the GUID to the element
    WSAddressingUtil.AddIdAttribute(addressNode, addressId);
    // Store the ID in a list for later signing
    addressIds.Add(addressId);
}

// Create a signature reference for each addressing element
foreach (string id in addressElemIds) {
    sigRef = new SignatureReference();
    sigRef.Uri = HashChar + id;
    msgSig.AddReference(sigRef);
}

// Add the message signature
security.Elements.Add(msgSig);
```

The MessageSignature object is then added to the security elements for processing by WSE.

Note the above code only specifies the signing configuration, but does not perform the actual signing operation. This will be performed by WSE after the SecureMessage method finishes.

5.4.2.4 Specify the encryption configuration

The service output filter can specify the encryption configuration for the response. This includes specify which key is used to encrypt, which parts of the message are encrypted, and how the encryption key is transported to the client.

An EncryptedData object is used to specify the encryption configuration. The EncryptedData object is created using the client token that was used to sign the corresponding request. This is the token that'll be used to perform the encryption. This is retrieved from the holder object (RequestState) that was set into the OperationState in section 5.2.1.2.

```
// Create the encrypted data object
EncryptedData encData = new EncryptedData(clientToken);

// Disable normal encryption of the body, so it can be controlled in code
encData.IncludeBodyElement = false;
```

The KeyInfo object is created using the key identifier reference type to satisfy criterion WS 6.2.7.2-1. This uses the subject key identifier of the certificate.

```
// Create a KeyInfo so it is possible to specify the key reference type
KeyInfo encKeyInfo = new KeyInfo();
encKeyInfo.AddClause(new SecurityTokenReference(serverToken,
    SecurityTokenReference.SerializationOptions.KeyIdentifier));
```

```
encData.EncryptedKey.KeyInfo = encKeyInfo;
```

The body and the signature are encrypted to satisfy criterion WS 6.2.4.2-1 and WS 6.2.4.2-2.

```
// Create an encryption reference for the body
EncryptionReference encRef = new EncryptionReference("#" + bodyElemId);
encRef.Type = XmlEncryption.TypeURI.Content;
encData.AddReference(encRef);

// Create an encryption reference for the signature
encRef = new EncryptionReference("#" + signatureElemId);
encRef.Type = XmlEncryption.TypeURI.Element;
encData.AddReference(encRef);

// Add the encrypted data
security.Elements.Add(encData);
}
}
```

The EncryptedData object is then added to the security elements for processing by WSE.

Note the above code only specifies the encryption configuration, but does not perform the actual encryption operation. This will be performed by WSE after the SecureMessage method finishes.

5.5 Create a custom exception class

The .NET *wsdl* tool does not create exception classes from the faults that are specified in the WSDL, therefore these exception classes must be created manually. The standard SOAP fault class does not contain a details section, so one needs to be added by extension.

5.5.1 Create the custom SOAP exception

A custom SOAP exception is created by extending the SoapException class:

```
using System;
using System.Web.Services.Protocols;
using System.Xml;
using System.Web.Services;
using System.Reflection;

using Nehta.Example.DS.Util;
using System.Collections.Generic;
using Nehta.Example.DS.Server.Util;
using System.Runtime.Remoting.Contexts;

namespace Nehta.Example.DS.Service.Exception {
    /// <summary>
    /// The InvalidIdFault class.
    /// </summary>
    public class InvalidIdFault : SoapException {
        private const string FaultName = "invalidIdFault";
        private const string FaultMessage = "Invalid document ID";
        private const string FaultDescription = "Has non-alphanumeric characters";
        private const string DocumentIdElementName = "documentId";
        private const string FaultDescriptionElementName = "faultDescription";
        private const string FaultDetailElementName = "Detail";
        private const string ServiceNamespacePrefix = "nsl";
        private const string SoapNamespacePrefix = "env";
        private const string FaultCode = "Sender";

        // ID that caused the exception to occur
        private string documentId;

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="id">ID that caused the error to occur.</param>
        public InvalidIdFault(string documentId)
            : base(FaultMessage, new XmlQualifiedName(FaultCode,
```

```

        Constants.SoapEnvelopeNamespace), null, CreateDetail(documentId))
    {
        this.documentId = documentId;
    }

    public string DocumentID {
        get { return this.documentId; }
        set { this.documentId = value; }
    }

```

The following creates the detail section of the fault:

```

private static XmlElement CreateDetail(string id) {
    string serviceNamespace = ServiceUtil.GetWebServiceNamespace();
    XmlDocument detailXmlDoc = new XmlDocument();

    // Create the detail element
    XmlElement detailElement = detailXmlDoc.CreateElement(SoapNamespacePrefix,
        FaultDetailElementName, Constants.SoapEnvelopeNamespace);

    // Create the fault element and add it to the detail element
    XmlElement rootElement = detailXmlDoc.CreateElement(ServiceNamespacePrefix,
        FaultName, serviceNamespace);
    detailElement.AppendChild(rootElement);

    // Create the description element
    XmlElement descElement = detailXmlDoc.CreateElement(ServiceNamespacePrefix,
        FaultDescriptionElementName, serviceNamespace);
    descElement.AppendChild(detailXmlDoc.CreateTextNode(FaultDescription));
    rootElement.AppendChild(descElement);

    // Create and add the 'id' element to the root
    XmlElement idElement = detailXmlDoc.CreateElement(ServiceNamespacePrefix,
        DocumentIdElementName, serviceNamespace);
    idElement.AppendChild(detailXmlDoc.CreateTextNode(id));
    rootElement.AppendChild(idElement);

    return detailElement;
}
}

```

The `GetWebServiceNamespace` method gets the namespace for the fault from the Web service implementation class; this prevents having to hardcode the namespace into the implementation of the exception.

```

public static string GetWebServiceNamespace() {
    object[] attrList = typeof(DSRService).GetCustomAttributes(
        typeof(WebServiceAttribute), false);
    WebServiceAttribute webServiceAttr = (WebServiceAttribute)attrList[0];
    return webServiceAttr.Namespace;
}

```

5.5.2 Throw the exception

The custom SOAP exception can then be thrown from a Web service method using a normal `throw` statement:

```

if (!IsValid(CheckStatus1.documentId)) {
    // The ID was not valid so throw an exception and inform the caller
    throw new InvalidIdFault(CheckStatus1.documentId);
}

```

5.6 Implement the Web service

The following steps describe how to implement a Web service:

1. Implement the service interface that was generated in section 5.1;
2. Add the `WebService` attribute to the Web service class.

The Web service is implemented by creating a class that extends the service interface and implements all the operations. The `WebService` attribute is added to the Web service class with the `Namespace` parameter set to the namespace of the Web service.

```
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Diagnostics;

using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Messaging;
using Microsoft.Web.Services3.Design;

using Nehta.Example.DS.Server.PolicyAssertion;
using Nehta.Example.DS.Server.Exception;
using Nehta.Example.DS.Server.Database;
using Nehta.Example.DS.Server.Util;

namespace Nehta.Example.DS.Server {
    [WebService(Namespace =
        "http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0")]
    [Policy(typeof(ServicePolicy))]
    [SoapActor("")]
    public class Service : IDischargeSummaryReceiverBinding {
        ...

        public pingResponse ping(ping ping1) {
            ...
        }

        public checkStatusResponse checkStatus(checkStatus checkStatus) {
            ...
        }

        public sendDischargeSummaryResponse sendDischargeSummary(
            sendDischargeSummary sendDischargeSummary) {
            ...
        }

        ...
    }
}
```

5.7 Configure the Web service

The following need to be configured for the Web service:

- WSE;
- Revocation mode;
- Encryption algorithms;
- Service policy;
- *Service.asmx* file.

5.7.1.1 WSE

WSE must be enabled on a Web service.

The following configuration from the *web.config* enables WSE:

```
<configuration>
  <configSections>
    <section name="microsoft.web.services3" type="..." />
  </configSections>
  <system.web>
    <webServices>
      <soapExtensionImporterTypes>
        <add
          type="Microsoft.Web.Services3.Description.WseExtensionImporter, ..." />
      </soapExtensionImporterTypes>
      <soapServerProtocolFactory type="Microsoft.Web.Services3.WseProtocolFactory,
        Microsoft.Web.Services3, Version=3.0.0.0 ... />
    </webServices>
  </system.web>
</configuration>
```

```

</webServices>
<compilation>
  <assemblies>
    <add assembly="Microsoft.Web.Services3, Version=3.0.0.0, ..." />
  </assemblies>
</compilation>
</system.web>
...
</configuration>

```

5.7.1.2 Revocation mode

The revocation mode determines if the certificates used in the security processing are checked for revocation. When doing testing, this should be disabled.

The following configuration is added to the *web.config* file:

```

<configuration>
  ...
  <microsoft.web.services3>
    <security>
      <x509 storeLocation="LocalComputer"
        revocationMode="NoCheck"
        verificationMode="TrustedPeopleOrChain" />
    </security>
  </microsoft.web.services3>
  ...
</configuration>

```

5.7.1.3 Encryption algorithms

The encryption algorithms need to set to satisfy criterion WS 6.2.6.1-1. The key algorithm is set to RSA15 and the session key algorithm is set to AES256.

The following configuration from the *web.config* file sets the encryption algorithms:

```

<configuration>
  <microsoft.web.services3>
    <security>
      ...
      <binarySecurityTokenManager>
        <add
          Type="Microsoft.Web.Services3.Security.Tokens.X509SecurityTokenManager,
          Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
          PublicKeyToken=31BF3856AD364E35" valueType="http://docs.oasis-
          open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3">
          <keyAlgorithm name="RSA15" />
          <sessionKeyAlgorithm name="AES256" />
        </add>
      </binarySecurityTokenManager>
    </security>
  </microsoft.web.services3>
</configuration>

```

5.7.1.4 Service policy

The `Microsoft.Web.Services3.PolicyAttribute` attribute is used to specify the custom security policy used for the Web service. The attribute is added to the service implementation class. The constructor has a single parameter which is a class that inherits from the `Microsoft.Web.Services3.Design.Policy` class and implements the policy. The policy contains the all assertions that are required.

The following is an example of `Microsoft.Web.Services3.Design.Policy` class:

```

internal class ServicePolicy : Policy {

```

```

public ServicePolicy() {
    ExampleDSPolicyAssertion serviceAssertion =
        new ExampleDSPolicyAssertion();
    string serviceCertId = ServiceUtil.GetServiceCertificateID();
    serviceAssertion.ServiceCertificateID = serviceCertId;

    this.Assertions.Add(serviceAssertion);
}
}

```

The constructor creates a service policy assertion object, sets the service certificate ID, and adds it to the list of assertions the policy uses. In this example, the policy only has the one service policy assertion.

The `Microsoft.Web.Services3.Policy` attribute is added to the Web service implementation class using the `ServicePolicy` class as the parameter:

```

[WebService(Namespace =
    "http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0")]
[Policy(typeof(ServicePolicy))]
[SoapActor("")]
public class Service : IDischargeSummaryReceiverBinding {
    ...
}

```

5.7.1.5 *Service.asmx* file

The `Class` attribute in the default `Service.asmx` file must be changed to match the full class name of the service.

```

<%@ WebService Language="C#" CodeBehind="~/App_Code/DSRService.cs"
    Class="Nehta.Example.DS.Service.DSRService" %>

```

5.8 Package and deploy the Web service

Once the Web service has been created and configured, it can be deployed using the in-built Visual Studio 2005 development server or to the Internet Information Services (IIS).

5.8.1 Development server

Visual Studio 2005 has an in-built development server that can be used to test and debug Web applications and Web services.

To deploy a Web service to the development server:

1. Right click the project from within the solution explorer and select "Set as StartUp Project".
2. Press F5 to start-up the development server and deploy the service. If a list of files is displayed in the browser, select the *Service.asmx* file.

To stop development server from running:

1. Right click on the system tray icon called "ASP.NET Development Server" and select "Stop".

5.8.2 Deploy with IIS

5.8.2.1 Create a compiled Web service (optional)

When deploying a service under IIS a virtual directory needs to be created. A virtual directory in IIS is a reference to a physical directory that can be on the local machine or a networked machine. The directory contains the Web service files, either as source files or compiled files. Generally source files are used for debugging and testing, and the compiled files for production systems.

To create a Web service binary distribution:

1. Select the Web service project within the solution explorer.
2. From the "Build" menu select "Publish Web Site".
3. Select the location to put the compiled Web site. This location can later be used as a virtual directory for IIS.
4. Select "OK" and the compiled Web service will be placed into the directory specified.

5.8.2.2 Create a virtual directory

One way to create a virtual directory is to use the IIS snap-in for the Microsoft Management Console (MMC).

To setup the MMC:

1. Start the MMC by selecting Run from the start menu and type *mmc* in the Open: combo box and press enter, or, by entering "mmc" within a command prompt window, this opens a blank Microsoft Management Console.
2. From the "File" menu, select "Add/Remove Snap-in".
3. Press the "Add..." button and select "Internet Information Services" from the list.
4. Select "Close" and then "OK" on the "Add/Remove Snap-in" dialog.
5. Save the MMC for later use.

To create a virtual directory:

1. Open the "Internet Information Services" MMC.
2. Expand the local computer node.
3. Expand the "Web Sites" node.
4. Right click on the "Default Web Site" node and select "New" and then "Virtual Directory..."
5. Select "Next" and then enter the alias for the Web site. The alias is used as part of the URL. For example, if the alias was "somealias", the URL for the Web service would be `http://somehostname[:port]/somealias/Service.asmx`
6. Select the directory where the Web service files are located. This can be compiled Web service that was created in Section 5.6.2.1 or the project directory which contains the source code.
7. Make sure the "Read" and "Run scripts" options are checked.
8. Select "Finish" and the virtual directory will be created.

Once the directory has been created, only the contents of the virtual directory need to be updated to redeploy the service. The virtual directory does not need to be re-created each time the code has changed.

5.8.2.3 Remove a virtual directory

When the virtual directory is no longer required, it can be removed from IIS.

To remove the virtual directory from IIS:

1. Open the "Internet Information Services" MMC.
2. Expand the local computer node.
3. Expand the "Web Sites" node.
4. Expand the "Default Web Site" node.
5. Right click on the name of the virtual directory and select "Delete".
6. Select "Yes" and the directory will be deleted.

Appendix A: References

- [JAXWS] Sun Microsystems, Java API for XML Web Services (JAX-WS), <https://jax-ws.dev.java.net>.
- [MSDNPA] Microsoft Development Network, How to: Create a Custom Policy Assertion that Secures SOAP Messages, 2007, <http://msdn2.microsoft.com/en-us/library/aa528788.aspx>
- [MSDNSE] Microsoft Development Network, The ASP Column: Using SOAP Extensions in ASP.NET, March 2004, <http://msdn.microsoft.com/en-us/magazine/cc164007.aspx>
- [NDS2006] NEHTA, National Discharge Summary: Data Content Specifications, version 1.0, 21 December 2006.
- [NIF2006] NEHTA, Interoperability Framework, version 1.0, 1 April 2006.
- [PING] Muus, The Story of the PING Program, <http://ftp.arl.mil/~mike/ping.html>.
- [TAIS2006] NEHTA, Technical Architecture for Implementing Services: Concepts and Patterns v1.0, 21 December 2006.
- [WCF] Microsoft, Windows Communication Foundation (WCF), <http://netfx3.com>.
- [WSE] Microsoft, Web Services Enhancements (WSE), <http://msdn2.microsoft.com/en-us/webservices/aa740663.aspx>.
- [WSP2008] NEHTA, *Web Services Profile v3.0*, 1 December 2008.
- [WSA2004] W3C, *Web Services Addressing (WS-Addressing)*, W3C Member Submission, 10 August 2004, <http://www.w3.org/ws/2004/08/addressing>.
- [WSAM2007] W3C, *Web Services Addressing 1.0 – Metadata*, W3C Recommendation, 4 September 2007, <http://www.w3.org/TR/2007/REC-ws-addr-metadata-20070904>.
- [WSPL2006] XmlSoap, *Web Services Policy Framework*, version 1.2, Draft, March 2006, <http://specs.xmlsoap.org/ws/2004/09/policy/ws-policy.pdf>.
- [WSPL2007] W3C, *Web Services Policy 1.5 – Framework*, W3C Recommendation, 4 September 2007, <http://www.w3.org/TR/2007/REC-ws-policy-20070904>.
- [WSSPL2005] XmlSoap, *Web Services Security Policy Language (WS-SecurityPolicy)*, Draft, July 2005, <http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf>.
- [WSSPL2007] OASIS, *WS-SecurityPolicy 1.2*, OASIS Standard, 1 July 2007, <http://docs.oasis-open.org/ws-sx/wssecuritypolicy/200702/ws-securitypolicy-1.2-spec-os.pdf>.

Appendix B: Installation

The following sections detail how to install the tested environment. All the software was installed on Microsoft Windows XP Professional SP2.

B.1 Visual Studio 2005

Visual Studio 2005 is an IDE used for developing .NET applications.

1. Run *setup.exe* from the *vs* directory on the installation DVD. This will install the Visual Studio 2005 onto the machine.

B.2 Web Services Enhancements 3.0 (WSE 3.0)

WSE is an add-on for .NET that enables the development of secure Web services and Web service clients.

1. Download *Microsoft WSE 3.0.msi*
URL:
<http://www.microsoft.com/downloads/details.aspx?familyid=018A09FD-3A74-43C5-8EC1-8D789091255D&displaylang=en>
2. Run the *Microsoft WSE 3.0.msi* file to start the installation wizard.
3. When prompted to select the "Setup Type", select the "Visual Studio Developer" option.
4. Once the installation wizard has completed, select "Finish" and WSE 3.0 will be ready to use.

B.3 Windows HTTP Services Certificate Configuration Tool

The *Windows HTTP Services Certificate Configuration Tool* is used for granting access to private keys from computer user accounts. This is necessary for a service when it requires the use of a private key to perform security operations.

1. Download *winhttpcertcfg.msi*
URL:
<http://www.microsoft.com/downloads/details.aspx?familyid=c42e27ac-3409-40e9-8667-c748e422833f&displaylang=en>
2. Run *winhttpcertcfg.msi* to install.
3. Add the installation directory of the tool to the PATH environment variable so it can be called from a command prompt.

B.4 Internet Information Services (IIS) (optional)

IIS can be used to host Web service created with WSE 3.0. Only install IIS when planning to make Web services accessible from outside the machine the services are being developed on. Use the Visual Studio 2005 development server to do testing within the local development machine.

1. Insert the Windows XP SP 2 CD/DVD into the drive.
2. Open the "Control Panel" and select "Add/Remove Programs".
3. From the left hand set of icons select "Add/Remove Windows Components".

4. On the "Windows Components Wizard" dialog, check the "Internet Information Services (IIS)" option and select "Next". IIS will then be installed.

ASP.NET needs to be installed into IIS before .NET applications can be run under IIS. When IIS is installed after Visual Studio, the ASP.NET components must be installed into IIS manually. To install ASP.NET use the ASP.NET registration command-line tool *aspnet_regiis.exe*. This is located in the *C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727* directory.

From a command window run the following to install ASP.NET into IIS:

```
aspnet_regiis -i
```

Appendix C: Key management

Windows stores certificates in the Windows certificate store and provides access to the security tokens from applications and services through an API. The store can also be accessed and managed using the Microsoft certificate MMC. The store is split into two locations, *CurrentUser* and *LocalComputer*. Each location is then split into a number of logical stores that hold security tokens with each store having a specific purpose.

User certificates with private keys are stored in the *CurrentUser* location within the *My* store, with *CA* certificates being placed in the *Trusted Root Certificate Authorities* store and other issued certificates in the *Other People* store. Service certificates are normally stored in the *LocalComputer* location within the *My* store with *CA* certificates being placed in the *Trusted Root Certificate Authorities* store.

C.1 Setting up the certificate MMC

The certificate MMC is used for managing certificates installed within the Microsoft certificate store.

1. Start the MMC by selecting Run from the start menu and type *mmc* in the Open: combo box and press enter, or, by entering "mmc" within a command prompt window, this opens a blank Microsoft Management Console.
2. From the "File" menu, select "Add/Remove Snap-in".
3. Press the "Add..." button and select "Certificates" from the list.
4. Select "My user account" and select "Finish", this will add a management snap-in that'll allow personal certificates to be managed.
5. Press the "Add..." button again and select "Certificates" from the list.
6. Select "Computer Account" and then select "Local computer:" on the select computer dialog.
7. Select "Close" and then "OK" on the "Add/Remove Snap-in" dialog.

C.2 Installing certificates for the client application

The client application requires the client's private key to sign the message, a service certificate for encrypting the message, and a CA (certificate authority) certificate for verification. The follow sections describe how to add the client certificates that are required by the client in order to secure and validate the message.

The following steps are required to setup the client application certificates:

1. Add the client certificate (that has a private key) which is used for signing.
2. Add the service certificate which is used for encrypting.
3. Add the CA.

C.2.1 Adding the client certificate

1. Open the certificate MMC.
2. Expand the "Certificates–Current User" tree by clicking on the plus sign.
3. Right click on the "Personal" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.
4. Select "Next" to go to the "File to Import".

5. On the "File to Import" dialog, select "Browse" then choose the PFX file that contains the client certificate and private key and select "Next".
6. Enter the password (if any) of the private key and select "Next".
7. Select "Next" again the finally select "Finished", the certificate will then appear within the "Certificates" folder under "Personal".

C.2.2 Adding the service certificate

1. Open the certificate MMC.
2. Expand the "Certificates – Current User" tree by left clicking on the plus sign.
3. Right click on the "Personal" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.
4. Select "Next" to go to the "File to Import".
5. On the "File to Import" dialog, select "Browse" then choose the service certificate and select "Next".
6. Select "Next" again the finally select "Finished", the certificate will then appear within the "Certificates" folder under "Personal".

C.2.3 Adding the CA

1. Open the certificate MMC.
2. Expand the "Certificates – Current User" tree by left clicking on the plus sign.
3. Right click on the "Trusted Root Certification Authorities" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.
4. Select "Next" to go to the "File to Import".
5. On the "File to Import" dialog, select "Browse" then choose the certificate file that is the CA and select "Next".
6. Select "Next" again and finally select "Finished", the certificate will then appear within the "Certificates" folder under "Trusted Root Certification Authorities".

C.3 Setting up the service certificates

The Web service requires a certificate with a private key to sign the message. A client certificate which would be used for encryption does not be specified since this can be extracted from the client request. The following sections describe how to add the certificates that are required by the service in order to secure and validate messages.

The following steps are required to setup the service certificates:

1. Add the service certificate (that has a private key) which is used for signing the response to the client.
 - a. Set the service certificate permissions.
2. Add the CA.

C.3.1 Adding the service certificate

1. Open the certificate MMC.
2. Expand the "Certificates (Local Computer)" tree by left clicking on the plus sign.

3. Right click on the "Personal" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.
4. Select "Next" to go to the "File to Import".
5. On the "File to Import" dialog, select "Browse" then choose the service certificate and select "Next".
6. Select "Next" again the finally select "Finished", the certificate will then appear within the "Certificates" folder under "Personal".

C.3.1.1 Set the service certificate permissions

For ASP.NET to be able to access the private key of the service certificate it must have the correct security permissions. The "Windows HTTP Service Certificate Configuration Tool" can be used to set the permissions on the certificate so that the "ASPNET" account has access to the private key.

To set the permissions on the service certificate:

1. Open a command prompt
2. Execute the following command:

```
winhttpcertcfg -g -c LOCAL_MACHINE\My -s <certificate name>
-a ASPNET
```

C.4 Adding the CA

1. Open the certificate MMC.
2. Expand the "Certificates (Local Computer)" tree by left clicking on the plus sign.
3. Right click on the "Trusted Root Certification Authorities" folder and select "All Tasks" and then "Import", the "Certificate Import Wizard" dialog should appear.
4. Select "Next" to go to the "File to Import"
5. On the "File to Import" dialog, select "Browse" then choose the certificate file that is the CA and select "Next".
6. Select "Next" again and finally select "Finished", the certificate will then appear within the "Certificates" folder under "Trusted Root Certification Authorities".

C.5 Pitfalls

When installing the security tokens into the Windows certificate store always use the certificate MMC import function. The certificate MMC allows the security tokens to be copied between locations and logical stores but this can cause problems when there's a private key.

Appendix D: Debugging

There are a number of ways to debug WSE applications. This appendix describes some possible debugging techniques which can be used:

1. Diagnostics configuration;
2. HTTP debugging proxy; and
3. Debugging a service that uses IIS.

D.1 Diagnostics configuration

When implementing and testing a client or Web service, message tracing can be enabled. The configuration for the message tracing can be added through the "WSE 3.0 Settings" dialog or by adding it to the *app.config* or *web.config* files directly.

To add the configuration:

1. Open the "WSE 3.0 Settings" dialog on the project.
2. Select the "Diagnostics" tab.
3. Click and enable "Enable Message Trace".
4. If detailed faults are required, select the "Send Detailed Error Information".

D.2 HTTP debugging proxy

A HTTP debugging proxy can be used to view all messages that are sent to and received from a client. This can be useful to see the raw SOAP messages in XML, and any HTTP headers that are sent with the message.

Once the HTTP debugging proxy has been installed and configured, the client must be configured to route messages through the debugging proxy. This can be done by setting the proxy object on the client proxy.

To configure the client to use the HTTP debugging proxy, within the client application code create a `WebProxy` object with the address and port of the proxy, and set the `Proxy` property on the client generated proxy class:

```
serviceInstance.Proxy = new WebProxy("http://<computer name>:8888", false);
```

Make sure to use the computer name where the proxy is running even if it's the local computer.

D.3 Debugging a service that uses IIS

To debug a service that's running under IIS:

1. Deploy the service to IIS.
2. From within Visual Studio 2005, select the "Debug" menu and then "Attach to Process...".
3. From the list of processes, select `aspnet_wp.exe` to connect to the ASP.NET process that runs the service. Now normal debugging operations can take place.