nehta

Example Technical Implementation of Interoperable Web services

WSE 3.0

Version 1.0 — 3 July 2007

Public release

National E-Health Transition Authority

National E-Health Transition Authority Ltd
Level 25
56 Pitt Street
Sydney, NSW, 2000
Australia.
www.nehta.gov.au

Disclaimer

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

Document Control

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

Copyright © 2007, NEHTA.

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

Table of contents

1 Introduction 1

1.1 Background 1
1.2 Purpose 1
1.3 Scope 1
1.4 Intended audience 1
1.5 Definitions, acronyms, abbreviations 1
1.5.1 Terminology 2
1.6 Style Conventions 2
1.7 Overview 2

2 Example service 4

2.1 Organisational 4
2.1.1 Testing 4
2.1.2 Sending discharge summaries 5
2.1.3 Checking discharge summary status 5
2.2 Informational 5
2.2.1 Discharge summary 5
2.2.2 Status 6
2.3 Technical 6
2.3.1 Informational attributes 6
2.3.2 Behavioural attributes 6
2.3.3 Non-functional attributes 11

3 Overview of WSE 3.0 12

3.1 General background 12
3.2 Technical overview 12
3.2.1 Message handling 13
3.3 Requirements 13
3.3.1 Client deployment 13
3.3.2 Web service deployment 14
3.3.3 Client development 14
3.3.4 Web service development 14
3.4 Platform used 14

4 Web service client 15

4.1 Generate the proxy from the WSDL files 15
4.2 Create a custom policy assertion 16
4.2.1 Create the policy assertion 16
4.2.2 Client output filter 17
4.2.3 Client input filter 20
4.3 Creating a SOAP extension 21
4.3.1 BeforeDeserialize: WS-Addressing conversion 23
4.3.2 AfterDeserialize: Setting DidUnderstand flag 24
4.4 Create the client application 25
4.4.1 Configure WSE 25
4.4.2 Call the proxy object 26

5 Web service 30

5.1 Generate Service Interface from the WSDL 30
5.2 Service custom policy assertion 30
5.2.1 Server input filter 31
5.2.2 Server output filter 33
5.3 Create a custom exception class 36
5.3.1 Create the custom exception 36
5.3.2 Throw the exception 37

5.4 Implementing the service ... 38
5.5 Configure the service ... 38
5.5.1 Configure the security policy ... 38
5.5.2 Configure the logical name of the service ... 39
5.5.3 Configure the location of the certificates ... 39
5.6 Packaging and deployment ... 39
5.6.1 Development server ... 39
5.6.2 Deploy with IIS ... 39

Appendix A: References ... 41

Appendix B: Installation ... 42
B.1 Visual Studio 2005 ... 42
B.2 Web Services Enhancements 3.0 (WSE 3.0) ... 42
B.3 Windows HTTP Services Certificate Configuration Tool ... 42
B.4 Internet Information Services (IIS) (optional) ... 42

Appendix C: Key management ... 44
C.1 Setting up the certificate MMC ... 44
C.2 Installing certificates for the client application ... 44
C.2.1 Adding the client certificate ... 44
C.2.2 Adding the server certificate ... 45
C.2.3 Adding the CA ... 45
C.3 Setting up the server certificates ... 45
C.3.1 Adding the server certificate ... 45
C.4 Adding the CA ... 46

Appendix D: Debugging ... 47
D.1 Diagnostics configuration ... 47
D.2 HTTP debugging proxy ... 47
D.3 Debugging a service that uses IIS ... 47

Document information

Change history

Version	Date	Comments
1.0	2007-07-03	Public release

This page is intentionally left blank.

1 Introduction

1.1 Background

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

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

1.2 Purpose

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

Its main purpose is to support the understanding and interpretation of the guidelines. However, it can also assist programmers who are learning how to use 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 NEHTA *Web Services Standards Profile* [WSSP2006] and the *Guidelines for Implementing Interoperable Web Services* [GIIWS2007]. The guidelines from [GIIWS2007] are referred to by their guideline number (e.g. "Guideline TG2").

1.5 Definitions, acronyms, abbreviations

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

1.6 Style Conventions

This document uses the following style conventions:

Italics	• Document titles • Program names, tool names • File names, directory paths • URLs
`Monospace`	• 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`	• Emphasis for within XML and code fragments
“Double quotes”	• Graphical user interface options

1.7 Overview

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

Chapter 3 provides a brief overview of 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 Java JAX-WS and WCF.)

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

2.1.2 Sending discharge summaries

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

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

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

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

2.1.3 Checking discharge summary status

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

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

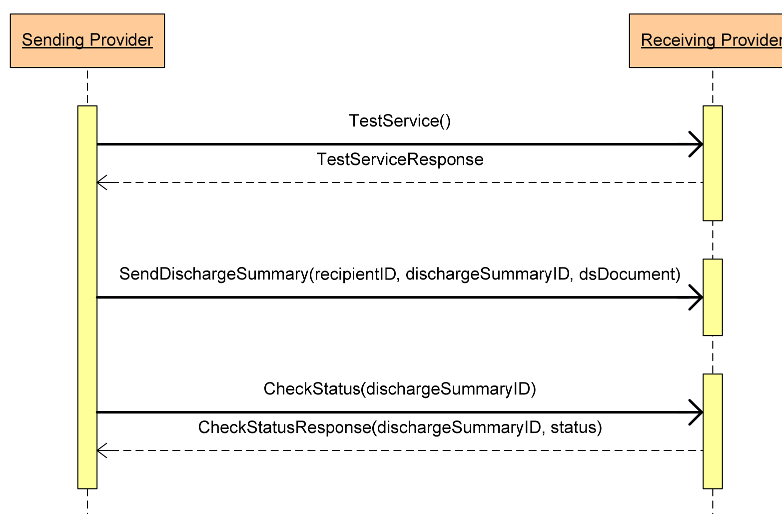

Figure 1: Example discharge summary business workflows

2.2 Informational

2.2.1 Discharge summary

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

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

2.2.2 Status

The possible status values for a discharge summary are:

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

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

2.3 Technical

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

2.3.1 Informational attributes

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

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

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

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

</xsd:schema>
```

2.3.2 Behavioural attributes

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

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

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

2.3.2.1 WSDL containing service interface information

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

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

2.3.2.1.1 Header

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

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

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

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

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

2.3.2.1.2 Types

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.2.1.3 Messages

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

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

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

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

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

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

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

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

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

2.3.2.1.4 *Port Type*

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

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

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

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

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

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

  </wsdl:portType>

</wsdl:definitions>
```

2.3.2.2 WSDL containing service instance information

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

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

2.3.2.2.1 *Header*

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

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

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

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

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

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

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

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

2.3.2.2.2 Binding

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

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

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

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

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

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

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

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

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

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

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

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

```
      <soap:operation style="document"

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

  </wsdl:binding>
```

2.3.2.2.3 Service

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

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

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

</wsdl:definitions>
```

2.3.3 Non-functional attributes

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

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

Conformance to these guidelines is not done through the WSDL. It is done through how the Web service client and Web service is implemented, which is the subject of chapters 4 and 5.

3 Overview of WSE 3.0

3.1 General background

The Web Service Enhancements (WSE) 3.0 for Microsoft .NET is a toolkit for building distributed applications. The toolkit consists of class libraries and supporting tools for those libraries.

Microsoft's .NET platform has basic support for Web services. WSE builds on top of the basic Web services to provide support for additional Web services standards. Of particular importance is WSE's support for WS-Security, which is an important part of NEHTA's recommended standards.

WSE 3.0 is designed to work with the Microsoft .NET Framework 2.0 and Visual Studio 2005. The previous release, WSE 2.0, is designed to work with the .NET Framework 1.1 and Visual Studio .NET 2003. Version 3.0 of the WSE toolkit is significantly different from version 2.0 and is not compatible. These older versions will not be covered in this document.

WSE 3.0 is the final release of WSE. Microsoft's Web services strategy is now focused on Windows Communications Foundation (WCF).

3.2 Technical overview

WSE 3.0 operates as a set of filters which intercept SOAP messages as they are being sent and received. When a message is sent, the filters add appropriate headers to the SOAP message. After the filtering, the combined SOAP message is sent across the network. When a message is received, the filters process the appropriate headers they understand. Once processed, the headers are removed from the SOAP message. After the filtering, the remaining SOAP message is passed on for normal processing under the .NET Framework.

The security filters are provided by WSE 3.0 to implement WS-Security. The outbound message filters are responsible for adding the security headers and transforming elements of the message that are being secured. The inbound filters are used to validate incoming messages and remove the processed security headers.

WSE includes a number of turnkey security assertions, which are a scenario based methodology for securing messages to allow predefined common security configurations to be used. The turnkey assertions support common security scenarios, such as using mutual certificate authentication and using username/password authentication. The toolkit also allows custom policy assertions to be created when the turnkey assertions are not sufficient. This requires implementing an assertion class and filter classes that specify how the WSE toolkit handles incoming and outgoing messages.

A Web service can be created from an existing WSDL or by writing code. When creating a Web service from a WSDL, WSE provides a WSDL utility program that is used to create the service interface that can then be implemented. This alleviates the need to write the service interface code from scratch and also assures it will work with the WSDL.

Both services and clients can use WSE for securing messages but they don't have to be used together. Service applications that are created in WSE can be hosted within IIS using SOAP with HTTP or the can be hosted outside IIS within an application and use SOAP with TCP.

3.2.1 Message handling

This section describes the hooks provided by Microsoft .NET 2.0 and WSE 3.0 for extending the behaviour of the toolkits. These hooks will be used in the implementation of the client and server.

3.2.1.1 SOAP extensions

Under .NET 2.0, SOAP extensions are a mechanism that allows SOAP messages to be inspected and modified.

There are four possible stages where a SOAP extension can intercept 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 across the network.
3. Before deserialisation: this stage allows manipulation of the XML of the incoming message, after it has been received from the network, 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, but before the processing application receives it.

The first and second stages apply to SOAP messages that are being sent. The third and fourth stages apply to SOAP messages that are being received.

The first and the last stages process allow manipulation of the message as objects. The second and third stages allow manipulation of the message in the serialised XML format (i.e. manipulated via the DOM).

3.2.1.2 Policy Assertions

Policy assertions are also a mechanism that allows SOAP messages to be inspected and modified. Policy assertions are a feature of WSE 3.0, whereas SOAP extensions are a feature of .NET 2.0. Normally, policy assertions are used by the WSE 3.0 framework to implement the security policies for the message (i.e. they perform the necessary signing and encryption operations, or decryption and signature verification operations).

A default set of turnkey policy assertions are provided with WSE 3.0. These implement basic signing and encryption scenarios. However, custom policy assertions can also be written to implement different scenarios.

Policy assertions can intercept a SOAP message before the message is sent, or before it is received by a processing application.

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. However, there are other ways to deploy a WSE 3.0 Web service that does not require IIS. These other ways 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 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 – Environment such as an IDE for creating .NET applications.

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 WSE 3.0 that conforms to the NEHTA *Guidelines for Implementing Interoperable Web Services* [GIIWS2007].

A WSE 3.0 client can be implemented using the following steps:

1. Generate the proxy from the WSDL files;

   These classes contain methods that correspond to the operations on the service.

2. Create a custom policy assertion.

   The policy assertion is used to secure and validate the SOAP messages. In this example, it is also used to add WS-Addressing 1.0 elements to the outgoing messages.

3. Create a SOAP extension.

   SOAP extensions are used to manipulate SOAP messages when they are sent and received. In this example, they are used to process the WS-Addressing headers.

4. Create the client application.

   The client application calls methods on the proxy classes to invoke operations on the service.

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

4.1 Generate the proxy from the WSDL files

The first step is to generate the proxy code for the client. The proxy takes care of: serialising the operation calls to SOAP requests that are sent to the Web service, and deserialising the SOAP responses from the Web service. The client invokes operations on the Web service by invoking methods on the proxy.

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 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 declaration.

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.
- `/protocol`: specifies what protocol to use. This is set to SOAP 1.2 to comply with the NEHTA *Web Services Standards Profile* [WSSP2006].
- `/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 `Example.DS.Client`.

The *wsewsdl3* 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.

4.2 Create a custom policy assertion

Policy assertions in WSE 3.0 are used to control how security is applied to the messages that a Web service client or Web service sends and receives. WSE 3.0 provides a number of scenario based turnkey policy assertions. However, the turnkey policy assertions provided with WSE 3.0 do not satisfy the Guidelines [GIIWS2007]. Therefore, a custom policy assertion needs to be created.

A custom policy assertion consists of an assertion class and a number of filter classes. This section describes the implementation of the assertion class and the filter classes. This class is then used in the client program, as described in section 4.4.2.4.

The following section describes the creation of the policy assertion class and the filter classes.

4.2.1 Create the policy assertion

A custom policy assertion class needs to be an extension of the `SecurityPolicyAssertion` class provided by WSE 3.0. This class defines a number of methods that the extending class overrides, the important ones being the four 'create' methods that return filter objects which perform the actual message processing.

Since this is a custom policy assertion class for the client (as opposed to the Web service), it only needs to provide the following two filters:

- Client output filter that processes the requests sent by the client; and
- Client input filter that processes the responses and faults that are received by the client.

The methods to create the other two filters (the server input filter and server output filter) can simply return `null`, because they are not needed in a client application.

This example custom policy assertion class contains two properties, `ClientToken` and `ServerToken`, to set the security credentials that the output filter will use to secure the SOAP message. Depending on the needs of the application, other approaches can be used to set these security credentials (e.g. set them in the class constructor, or hard code them in the filters).

The example 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 Example.DS.PolicyAssertion.Client {
  public class ExampleDSClientPolicyAssertion : SecurityPolicyAssertion {
    private SecurityToken m_clientToken;
    private SecurityToken m_serverToken;

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

    // Get and set the server token used for encrypting
    public SecurityToken ServerToken {
      get { return m_serverToken; }
      set { m_serverToken = value; }
    }

    // Filter creation methods
```

```
    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; }

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

Further information on the creation of custom policy assertions can be found in [MSDN2007].

4.2.2 Client output filter

The client output filter processes the request messages when they are sent from the client.

In this example, the filter performs three tasks:

- Create and add the WS-Addressing 1.0 headers to the outgoing SOAP message. This is because WSE 3.0 has implemented an older 2004 draft version of the WS-Addressing specification, which uses elements from a different namespace from what the standard WS-Addressing 1.0 uses. Therefore, this filter will be used to explicitly add the WS-Addressing 1.0 headers to satisfy Guideline TG14.
- Identify the parts of the message that need to be signed to satisfy Guideline TG9, TG10, and TG11.
- Identify the parts of the message that need to be encrypted to satisfy Guideline TG9, TG10, and TG11.

4.2.2.1 Create the output filter class

The example client output filter will be called `ClientOutputFilter` in the namespace of `Example.DS.PolicyAssertion.Client.Filter`.

The filter class returned by the `CreateClientOutputFilter` method of the custom policy assertion must return an instance of the `ClientOutputFilter` class. The `ClientOutputFilter` class inherits from `SendSecurityFilter` and needs to override the `SecureMessage` method.

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

  class ClientOutputFilter : SendSecurityFilter {
    private SecurityToken m_clientToken;
    private SecurityToken m_serverToken;

    public ClientOutputFilter(
      ExampleDSClientPolicyAssertion clientAssertion, FilterCreationContext fcc)
      : base(clientAssertion.ServiceActor, true) {
      m_clientToken = clientAssertion.ClientToken;
      m_serverToken = clientAssertion.ServerToken;
    }

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

The first thing the `SecureMessage` method does is add a `wsu:ID` attribute (where `wsu` is the WS-Security utility namespace) to the SOAP body element. This `wsu:ID` will be used later on to identify the body of the SOAP message for signing and encrypting.

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

4.2.2.2 WS-Addressing 1.0

The following code then creates and adds the WS-Addressing 1.0 headers to the SOAP message. It also sets a `wsu:ID` attribute on all of the elements added, so they can be referred to during signing.

It uses the `ExampleDSUtil.ConvertToWSAddressing10` method to create the headers based on the existing draft WS-Addressing headers, and then adds them to the SOAP message.

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

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

The following code converts the 2004 addressing elements to the WS-Addressing 1.0 namespace. This is done by copying the existing elements (that were added by WSE) into elements with the new addressing namespace:

```
public static void ConvertToWSAddressing10(XmlNode soapMessage,
  bool deleteExisting)
{
  // Create the namespace manager
  XmlNamespaceManager xnm =
    new XmlNamespaceManager(soapMessage.OwnerDocument.NameTable);
  xnm.AddNamespace(SoapEnvelopePrefix,
    Constants.SoapEnvelopeNamespace);
  xnm.AddNamespace(WSAddressing2004Prefix,
    Constants.WSAddressing2004Namespace);

  // Get the header node
  XmlNode headerNode =
    GetXmlNode(soapMessage, SoapHeaderElementXPath, xnm);

  // Find the old addressing nodes
  XmlNodeList oldAddressingNodes =
    headerNode.SelectNodes(WSAddressing2004ElementsXPath, xnm);
  if (oldAddressingNodes.Count == 0) {
    throw new Exception("No 2004 addressing nodes 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 = ChangeNodeNamespace(oldAddrNode,
        WSAddressing10Prefix, Constants.WSAddressing10Namespace,
        true, false);

      headerNode.AppendChild(cloneNode);
    }

    if (deleteExisting) {
      foreach (XmlNode delNode in existingNodes) {
        headerNode.RemoveChild(delNode);
      }
    }
```

```
        }
    }
```

The GetXmlNode method is used to retrieve a node from the SOAP message using an XPath statement.

The ChangeNodeNamespace method is used to change the namespace of a node from one to another. This is necessary when copying the elements from the older draft addressing namespace to the new WS-Addressing 1.0 namespace. A new element is created with the same local name as the source element but in the WS-Addressing 1.0 namespace.

The GetWSAddressing10Elements method returns a list of nodes that are in the WS-Addressing 1.0 namespace using an XPath statement. This is needed so that an ID attribute can be added to each addressing element which can then be later used as a signing reference to satisfy Guideline TG9.

4.2.2.3 Signing

The following code then indicates which parts of the SOAP message need to be signed. It creates a MessageSignature object and populates it with the signature references.

Notice that it specifies that the client token that will be used to perform the signing.

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

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

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

// 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 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.

4.2.2.4 Encryption

The following code then indicates which parts of the SOAP message need to be encrypted. It creates an EncryptedData object, and adds it to the security object that was passed to the filter.

Notice that it specifies the server token that will be used to perform the encryption.

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

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

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

      // 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.

4.2.3 Client input filter

The client input filter processes the response and fault messages when they are received by the client.

In this example, the filter performs one task:

- Check for signature and encryption

4.2.3.1 Create the input filter class

The example client input filter will be called `ClientInputFilter` in the namespace of `Example.DS.PolicyAssertion.Client.Filter`.

The filter class returned by the `CreateClientInputFilter` method of the custom policy assertion must return an instance of the `ClientInputFilter` class. The `ClientInputFilter` class inherits from `ReceiveSecurityFilter` and needs to override the `ValidateMessageSecurity` method.

```
using System;

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

namespace 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)
    {
```

4.2.3.2 Check for signature and encryption

This filter simply checks that the message has been signed and encrypted. If it has not been signed, or it has not been encrypted, an exception is raised.

```
    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("Message must be signed");
    }

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

  } // ValidateMessageSecurity

 }
}
```

The actual signature validation operation, comparing the checksum to the message content and verifying the integrity of the signature, is handled by WSE. The check performed in this filter ensures that there is a signature in the message; otherwise WSE will not raise an exception if an unsigned message is received.

Similarly, the actual decryption operation is handled by WSE. The check performed in this filter ensures that the message is encrypted; otherwise WSE will not raise an exception if an unencrypted message is received.

The actions performed by this filter are kept simple for this example. In a real application, this filter could perform more sophisticated processing. For example, checking that the correct parts of the message are being signed and encrypted, checking that the acceptable mechanisms are being used for signing and encrypting, and checking the credentials being used for signing and encrypting.

4.3 Creating a SOAP extension

SOAP extensions can be used to inspect and/or modify SOAP messages at various stages during the message processing pipeline. They are similar to policy assertions in that they intercept the SOAP message as it is being sent or received. However, they differ from policy assertions, because they are they allow more low level control. This is because SOAP extensions are a feature of the Web services implementation in the .NET framework (whereas policy assertions are a feature of WSE 3.0). This low level control is most apparent in the different stages where SOAP extensions can be interposed.

There are four stages where SOAP extensions can be used. When a message is being sent, SOAP extensions can be interposed before the SOAP message is

serialised into XML or after serialisation. When a message is being received, SOAP extensions can be interposed before the SOAP message is deserialised from XML or after deserialisation. Where the SOAP extension is interposed will depend on whether the inspection or modification needs to be done on the XML encoded form or the objectified form of the message.

In this example, a SOAP extension will be used to convert the WS-Addressing 1.0 headers of incoming messages on the client and server to the draft WS-Addressing version WSE expects. As described previously, WSE 3.0 implements an older 2004 draft version of WS-Addressing instead of WS-Addressing 1.0. These two versions are not compatible, because their elements are in different XML namespaces. To conform to the NEHTA standards, WS-Addressing 1.0 must be used.

The example SOAP extension will be a class called `AddressSoapExtension` created in the `Example.DS.AddressSoapExtension` namespace. The important part of the class is the `ProcessMessage` method, which gets called with the message as its argument at the various stages of the SOAP message processing pipeline.

```
namespace Example.DS.AddressSoapExtension {

  public class AddressSoapExtension : SoapExtension {
    private Stream m_wireStream;
    private Stream m_appStream;

    public override Stream ChainStream(Stream stream) {
      m_wireStream = stream;
      m_appStream = new MemoryStream();
      return m_appStream;
    }

    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:
          // used by server: do nothing
          break;

        case SoapMessageStage.AfterSerialize:
          // used by server
          m_appStream.Position = 0;
          Copy(m_appStream, m_wireStream);
          break;

        case SoapMessageStage.BeforeDeserialize:
          AddAddressingElements(message); // see description below
          break;

        case SoapMessageStage.AfterDeserialize:
          SetAddressHeadersDidUnderstand(message); // see description below
          break;
      }
    }

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

The SOAP extension:

- Converts WS-Addressing elements in the `BeforeDeserialize` stage; and
- Sets the `DidUnderstand` flag of the addressing headers in the `AfterDeserialize` stage.

4.3.1 BeforeDeserialize: WS-Addressing conversion

The `BeforeDeserialize` stage allows manipulation of the XML messages that are received.

This example will convert the WS-Addressing 1.0 elements into draft 2004 WS-Addressing elements that WSE 3.0 can understand. This conversion is done in the SOAP extension because it can't be done in the custom policy assertion as WSE 3.0 would reject the SOAP message if the 2004 WS-Addressing `Action` header could not be found. However, WSE 3.0 checks for the draft 2004 WS-Addressing `Action` header before the custom policy assertion is used. Therefore, the conversion of the WS-Addressing 1.0 `Action` header into the draft 2004 WS-Addressing `Action` header must occur before that check is done. This is why it must be done in the `BeforeDeserialize` stage of the SOAP extension.

The function the example uses to do the conversion is called `AddAddressingElements`.

```
private void AddAddressingElements(SoapMessage message) {
  StreamReader readStr = new StreamReader(m_wireStream);
  StreamWriter writeStr = new StreamWriter(m_appStream);
  string soapMsg = readStr.ReadToEnd();

  // Ignore the message if it is empty
  if (soapMsg == null || soapMsg.Length == 0) {
    return;
  }

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

  // Do the conversion
  ConvertToWSAddressing2004(soapXmlDoc.DocumentElement, true);

  // Serialise the DOM back out
  soapMsg = soapXmlDoc.InnerXml;
  writeStr.Write(soapMsg);
  writeStr.Flush();

  m_appStream.Position = 0;
}
```

It uses the `ConvertToWSAddressing2004` function which copies the elements from the WS-Addressing 1.0 namespace to the draft WS-Addressing namespace.

```
    public static void ConvertToWSAddressing2004(XmlNode soapMessage,
      bool deleteExisting)
    {
      // Setup the namespace manager
      XmlNamespaceManager xnm =
        new XmlNamespaceManager(soapMessage.OwnerDocument.NameTable);
      xnm.AddNamespace(SoapEnvelopePrefix,
        Constants.SoapEnvelopeNamespace);
      xnm.AddNamespace(WSAddressing10Prefix,
        Constants.WSAddressing10Namespace);
      xnm.AddNamespace(WSAddressing2004Prefix,
        Constants.WSAddressing2004Namespace);

      // Find the header node
      XmlNode headerNode =
        GetXmlNode(soapMessage, SoapHeaderElementXPath, xnm);
```

```
      // Get the existing 2004 addressing nodes
      XmlNodeList oldAddressingNodeList =
        headerNode.SelectNodes(WSAddressing2004ElementsXPath, xnm);

      if (oldAddressingNodeList.Count == 0) {
        // Get the existing 2005 address nodes that will be copied
        XmlNodeList addressingNodeList =
          headerNode.SelectNodes(WSAddressing10ElementsXPath, 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 = ChangeNodeNamespace(addrNode,
              WSAddressing2004Prefix, Constants.WSAddressing2004Namespace,
              true, false);

            headerNode.AppendChild(clone);
          }

          // Check if a 'To' element exists and if there isn't add
          // one using the W3 anonymous namespace
          XmlNodeList toNodeList =
            headerNode.SelectNodes(WSAddressing10ToElementXPath, xnm);

          if (toNodeList.Count == 0) {
            // No 'To' element was found so add one
            XmlElement to = soapMessage.OwnerDocument.CreateElement(
              WSAddressing2004Prefix,
              Constants.WSAddressingToElementName,
              Constants.WSAddressing2004Namespace);
            to.AppendChild(soapMessage.OwnerDocument.CreateTextNode(
              Constants.AnonymousNamespace2004));

            headerNode.AppendChild(to);
          }

          if (deleteExisting) {
            // Delete the existing nodes
            foreach (XmlNode delNode in existingNodes) {
              headerNode.RemoveChild(delNode);
            }
          }
        }
      }
    }
```

The `GetXmlNode` method is used to retrieve a node from the SOAP message using an XPath statement.

The `ChangeNodeNamespace` method is used to change the namespace of a node from one to another. This is necessary when copying the elements from the older addressing namespace to the new WS-Addressing 1.0 namespace.

4.3.2 AfterDeserialize: Setting DidUnderstand flag

The `AfterDeserialize` stage allows manipulation of the received message in object form.

This example will set the `DidUnderstand` flag for the WS-Addressing 1.0 headers to true. These headers are usually received with the `mustUnderstand` attribute set to true. Therefore, if the client does not indicate that it understands them, an exception will be raised by WSE when it tries to processes them.

```
    // 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 2005 addressing namespace
          if (suh.Element.NamespaceURI.Equals(
              Constants.WSAddressing10Namespace,
              StringComparison.OrdinalIgnoreCase)) {
            sh.DidUnderstand = true;
          }
        }
      }
    }
```

4.4 Create the client application

For the client application to invoke an operation on a service instance, it uses the proxy classes, the custom policy assertion, the SOAP extension, and the following steps.

5. Configuring WSE:
   a. Enable WSE 3.0;
   b. Configure certificate repository;
   c. Configure SOAP extension.
6. Calling the proxy object:
   a. Creating a proxy object;
   b. Setting the Web proxy address (optional);
   c. Set WS-Addressing fields and destination;
   d. Set the security policy and security tokens; and
   e. Make a call to the service.

4.4.1 Configure WSE

The WSE 3.0 configuration is stored in the standard application configuration file called *app.config*. The policy assertion configuration is usually stored in a policy cache file called *wse3policyCache.config*. The *app.config* file can be edited using the WSE settings dialog, which can be accessed by right-clicking the project name within Visual Studio and selecting "WSE Settings 3.0...". Since the *app.config* file is in XML, it can also be edited with a text editor.

The instructions below will describe both the steps using the WSE settings dialog as well as what changes are being made to the XML file. Since the client policy assertion does not use a configuration file it will be configured in code. This is detailed in the section 4.4.2 "Calling the proxy object".

4.4.1.1 Enable WSE 3.0

The first item to configure is the enabling of WSE:

1. Right click on the client project and select "WSE Settings 3.0...";
2. Select the "General" tab; and
3. Select "Enable this project for Web Service Enhancements".

This will add the appropriate configuration section to the *app.config* file.

4.4.1.2 Configure the certificate repository

The second item to configure is the location of the certificate repository. When WSE tries to validate incoming messages that have been secured, it needs to

look up the Windows certificate repository to retrieve the certificates it needs. In particular, since this is the client that processes response messages, it needs the encrypting key for the client (to decrypt the message) and the signing certificate of the server (to verify the integrity of the message's signature). WSE looks in the certificate repository for this key and certificate.

1. Right click on the client project and select "WSE Settings 3.0…".
2. Select the "Security Tab".
3. Change the "Store location" to "CurrentUser". This instructs WSE to look in the "CurrentUser" store and not the "LocalComputer" store when it tries to find security tokens to validate an incoming message. Note that this store was chosen for the example and can be changed depending on where the security tokens are stored.
4. Change the "Revocation Mode" to "NoCheck". Since no revocation checking will be done in this example, this feature is disabled.
5. Click the "OK" button to save the settings to the *app.config* file.

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

```
<configuration>
  …
  <microsoft.web.services3>
    <security>
      <x509 storeLocation="CurrentUser" revocationMode="NoCheck" />
    </security>
  </microsoft.web.services3>
  …
</configuration>
```

4.4.1.3 Configure the SOAP extension

The third item to configure is the SOAP extension. This tells the application to use the SOAP extension class that was created in section 4.3. Since the SOAP extension is separate from WSE, it must be configured manually.

1. Open the client project within the example solution.
2. Open the *app.config* file.
3. Insert the SOAP extension configuration into the appropriate part of the application configuration file and save it.

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

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

Note that the SOAP extension class name and assembly name have been chosen for this example and can be set to anything.

4.4.2 Call the proxy object

4.4.2.1 Create a proxy object

The first step is to create an instance of the proxy class that was generated by the *wsewsdl3* tool in section 4.1.

In this example, the proxy object will be assigned to a variable called serviceInstance.

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

4.4.2.2 Set the Web proxy address (optional)

If the client program is running behind a Web proxy, then the address of the Web proxy needs to be configured. This is done by creating a `WebProxy` object and setting it to the `Proxy` member 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);
```

4.4.2.3 Set the WS-Addressing Fields and destination

The WS-Addressing `From` field is set to a logical name for the client, to satisfy Guideline TG14.

```
// Set "From" to logical name of this client
String clientLogicalName = "urn:dsr:wse_client";

serviceInstance.RequestSoapContext.Addressing.From =
  new Addressing.From(new Address(new Uri(clientLogicalName)));
```

The WS-Addressing `To` field is set to the logical name of the destination. Additionally, the URL address for the service instance is set as the `Via` address (the second parameter to the constructor of `EndpointReference`).

In the most simple cases, the `Via` address URL is the same as the address for the entity referred to by the logical name of the destination. However, if intermediaries are being used, the `Via` address URL will be that of the intermediary. The intermediary is supposed to then forward the message to the entity referred to by the logical name of the destination (perhaps going through other intermediaries).

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

4.4.2.4 Set the Security Policy

Configuring security on the client side requires calling the `SetPolicy` method on the generated client proxy class. The `SetPolicy` methods takes either a string parameter with the name of the policy cache file (this method has not been used in this example), or a class that extends `Policy`. Create a `Policy` object and then add the custom assertion to it.

```
…

ExampleDSClientPolicyAssertion clientAssertion =
  new ExampleDSClientPolicyAssertion();
```

The security token that will be used by the client to sign the message needs to be set on the client policy assertion. To set the security token, use the `X509SecurityToken` class to retrieve it from the Windows certificate store and

then use the `ClientToken` property to set it. The constructor for the `X509TokenProvider` class takes the store name, store location, and subject as parameters:

```
clientAssertion.ClientToken = X509TokenProvider.CreateToken(
  StoreLocation.CurrentUser, StoreName.My, clientCertificateId);
```

The security token that will be used by the client to encrypt the message needs to be set on client policy assertion. To set the security token use the `X509SecurityToken` class to retrieve it from the Windows certificate store and then use the `ServerToken` property to set it:

```
clientAssertion.ServerToken = X509TokenProvider.CreateToken(
  StoreLocation.CurrentUser, StoreName.AddressBook, serviceCertificateId);
```

Create the `Policy` object and add the client policy assertion:

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

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

4.4.2.5 Make a call to the service

Finally, the operation is invoked by calling the corresponding method on the proxy object.

4.4.2.5.1 Ping

This operation is the simplest to invoke because it requires no parameters and returns no results. Simply create a new `Ping` object (to represent the empty set of parameters) and call the `Ping` method on the client proxy object.

```
try
{
  serviceInstance.Ping(new Ping());
}
catch (…) {
  …
}
```

4.4.2.5.2 Send discharge summary

Invoking the send discharge summary operation involves:

1. Creating a `DischargeSummaryType` object to represent the discharge summary document;
2. Creating a `SendDischargeSummary` object to contain the parameters (i.e. the discharge summary and an identifier for it) for the operation; and
3. Calling the `SendDischargeSummary` method on the client proxy object.

```
// Create the discharge summary document itself
DischargeSummaryType ds = new DischargeSummaryType();
ds.Notes = "This is an example discharge summary";
ds.DocumentId = "someid";

// Create and set parameters
SendDischargeSummary sds = new SendDischargeSummary();
sds.Document = ds;

// Invoke the operation using the client stub "serviceInstance"
try {
  serviceInstance.SendDischargeSummary(sds);
}
catch (…) {
  …
}
```

4.4.2.5.3 *Check status*

Invoking the check status operation is similar to the other operations, except that this operation returns a result. This involves:

1. Creating a `CheckStatus` object to contain the parameters for the operation;
2. Calling the `CheckStatus` method on the client proxy object and storing the response in the `CheckStatusResponse` object; and
3. Processing the response 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 (…) {
  …
}
```

4.4.2.5.4 *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 WSE 3.0. The aim of a Web service is to provide a service instance that makes available service operations for clients to invoke.

The steps are:

1. Generate service interface from the WSDL;
2. Create custom policy assertion;
3. Create custom exception class;
4. Implement the service;
5. Configure the service; and
6. Deploy the service.

5.1 Generate Service Interface from the WSDL

The service interface classes correspond to the service and its operations, and are used as the basis for implementing the service.

The service interface classes are generated from the WSDL 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 the NEHTA *Web Services Standards Profile* [WSSP2006].
- /serviceInterface: indicates that a service interface should be generated.

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 5.4.

5.2 Service custom policy assertion

Policy assertions in WSE 3.0 are used to control how security is applied to the messages that the program sends and receives. They were described in section 4.2 for the example client; in this section, they are used for the example Web service.

Since this is a custom policy assertion class for the server, only these two filters are required:

- Server input filter that processes the requests received by the server; and
- Server output filter that processes the responses and faults sent by the server.

The methods to create the other two filters (client input filter and client output filter) can simply return `null`, because they are not used by WSE in a service.

The example policy assertion class is shown below.

```
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 Example.DS.PolicyAssertion.Server.Filter;


namespace Example.DS.PolicyAssertion.Server {
  public class ExampleDSServerPolicyAssertion : SecurityPolicyAssertion {
    private string m_serviceCertificateID;

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

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

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

    public override SoapFilter CreateServiceInputFilter(FilterCreationContext
context) {
      return new ServerInputFilter(this, context);
    }

    public override SoapFilter CreateServiceOutputFilter(FilterCreationContext
context) {
      return new ServerOutputFilter(this, context);
    }
  }
}
```

The class also has a property for getting and setting the service certificate ID. This is used for signing the message when a response is being sent.

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

5.2.1 Server input filter

The server input filter processes the request messages when they are received by the server.

In this example, the filter performs two tasks:

- Check for signature and encryption; and
- Save information needed to generate the response.

5.2.1.1 Create the server input filter class

To implement an input filter, extend the `ReceiveSecurityFilter` class and override the `ValidateMessageSecurity` method:

```
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 Example.DS.Util;


namespace Example.DS.PolicyAssertion.Server {
```

```
class ServerInputFilter : ReceiveSecurityFilter {

  public ServerInputFilter(ExampleDSPolicyAssertion exampleAssertion,
    FilterCreationContext fcc)
    : base(exampleAssertion.ServiceActor, false)
  {
    // constructor, do nothing
  }

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

5.2.1.2 Check for signature and encryption

As a basic check, this service input filter will check that the request message has been signed and encrypted.

```
    // Extract the security tokens

    SecurityToken clientToken = null;
    SecurityToken serverToken = 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 (serverToken != null) {
          throw new PolicyAssertionException("More than one encryption found");
        }
        else {
          EncryptedData encData = securityElem as EncryptedData;
          serverToken = encData.SecurityToken;
        }
      }
    }

    // Check if the message contains a signature and if does not throw an
    // exception
    if (clientToken == null) {
      throw new PolicyAssertionException("Message was not signed");
    }

    // Check if the message has encryption and if does not throw an
    // exception
    if (serverToken == null) {
      throw new PolicyAssertionException("Message was not encrypted");
    }
```

5.2.1.3 Save information needed to generate the response

The server will require the certificate for the service requestor, so that it can encrypt the response or fault message.

It also requires the value of the WS-Addressing `Action` from the request, so that the value for the WS-Addressing `Action` response can be generated.

A `RequestState` class has been created to contain this information. A `RequestState` object is created and added to the operation state within the input filter. The class stores the service requestor's certificate and the value of the WS-Addressing `Action`.

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

The `RequestState` class which is used to pass state from the input filter to the output filter:

```
using System;

using Microsoft.Web.Services3.Security.Tokens;

namespace Example.DS.PolicyAssertion.Server {
  class RequestState {
    private SecurityToken m_clientToken;
    private string m_actionValue;

    public RequestState(SecurityToken clientToken, string actionVal) {
      m_clientToken = clientToken;
      m_actionValue = actionVal;
    }

    public SecurityToken ClientToken {
      get { return m_clientToken; }
      set { m_clientToken = value; }
    }

    public string ActionValue {
      get { return m_actionValue; }
      set { m_actionValue = value; }
    }
  }
}
```

5.2.2 Server output filter

The server output filter processes the response and fault messages before they are sent to the client.

In this example, the filter performs three tasks:

- Modify the outgoing value of the WS-Addressing `Action` element. When WSE specifies the action value in the `Action` element of the addressing on outgoing messages from the server, the default behaviour is to append "Response" onto the incoming `Action` value from the client. This "Response" text needs to be removed and replaced with "Request" as per guideline TG7. Also WSE uses a fault action value on outgoing messages from the server that does not match the guidelines, so this case needs to be handled as well.
- Add the WS-Addressing 1.0 elements to the header of the message. This is because WSE 3.0 has implemented a draft version of WS-Addressing, which uses elements from a different namespace from what WS-Addressing 1.0 uses. Therefore, this filter will need to manually add the WS-Addressing 1.0 headers to satisfy Guideline TG14.
- Specify what needs to be signed and encrypted, as required by Guideline TG9, TG10, and TG11.

5.2.2.1 Create the server output filter class

To implement an output filter, extend the `SendSecurityFilter` class and override the `SecureMessage` method:

```
namespace ExampleDS.PolicyAssertion.Server {

  class ServerOutputFilter : SendSecurityFilter {

    public ServerOutputFilter(ExampleDSPolicyAssertion serverAssertion,
      FilterCreationContext fcc)
      : base(serverAssertion.ServiceActor, false)
    {
    }

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

Retrieve the token that was saved from the request and the `Action` value that was in the request message:

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

SecurityToken clientToken = requestState.ClientToken;
SecurityToken serviceToken =
  SecurityUtil.GetServiceToken(m_serviceCertificateID);
string actionValue = requestState.ActionValue;
```

The `GetServiceToken` method uses the service certificate ID to retrieve the service certificate from the Windows certificate store.

Generate and add a `wsu:ID` to the body element so it can be referenced as a target for signing and encrypting.

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

5.2.2.2 Modify Action

The WS-Addressing `Action` value that is created by a WSE Web service in a response messages does not match Guideline TG7 of the guidelines document and must be modified.

The `ExampleDSUtil.UpdateActionElement` method is used to modify the action value:

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

The following is an example implementation of the `UpdateActionElement` method that is used to modify the action value of outgoing messages from a WSE Web service:

```
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, FaultDetailElementXPath, 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;
  }
}
```

5.2.2.3 WS-Addressing

Add the WS-Addressing 1.0 headers:

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

Add a `wsu:ID` to each of the WS-Addressing elements and store them in a list so they can be used as signing targets.

```
// Create an ID for each address element add it to the element
List<string> addressIds = new List<string>();
XmlNodeList addressNodes =
  ExampleDSUtil.GetWSAddressing10Elements(envelope.DocumentElement);
foreach (XmlNode addressNode in addressNodes) {
  string addressId = Guid.NewGuid().ToString();
  ExampleDSUtil.AddIdAttribute(addressNode, addressId);
  addressIds.Add(addressId);
}
```

5.2.2.4 Signing

Create a `MessageSignature` object and add signature references for each signature target. Then add the created `MessageSignature` object to the security object that is passed to the method:

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

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

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

// 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 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);
```

5.2.2.5 Encryption

Create an `EncryptedData` object and add encryption references for each encryption target. Then add the created `EncryptedData` object to the security object that is passed to the method:

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

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

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

      // 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);
    }
  }
}
```

5.3 Create a custom exception class

The WSDL specifies operations and the messages and faults these operations use. 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 this needs to be added with a custom SOAP fault class that specifies a detail section.

5.3.1 Create the custom exception

Extend the `SoapException` class and add implementation that adds the detail section:

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

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


namespace Example.DS.Server.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 = "ns1";
    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; }
    }
```

Create the detail section:

```
    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 following method gets the namespace of the fault from the service namespace, 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;
}
```

5.3.2 Throw the exception

The custom exception can then be thrown from code using a normal `throw` statement:

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

5.4 Implementing the service

Using the generated interface, implement all the methods that comprise the service. The following steps describe how to create a service implementation from the service interface:

1. First create a class that implements the service interface that was generated in section 5.1 using the *wsdl* tool.
2. Add the `WebService` attribute to the class definition and specify the namespace of the service.
3. Implement all the methods in the service interface.
4. Change the `Class` attribute in the *Service.asmx* file to match the full type name of the 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 Example.DS.PolicyAssertion.Server;
using Example.DS.Server.Exception;
using Example.DS.Server.Database;
using Example.DS.Server.Util;


namespace ExampleDS.Server {
  [WebService(Namespace =
    "http://example.com/ns/2007/DischargeSummaryReceiver/v1")]
  [Policy(typeof(ServicePolicy))]
  [SoapActor("urn:dsr:wse_server")]
  public class Service : IDischargeSummaryReceiverBinding {
    …

    public PingResponse Ping(Ping Ping1) {
      PingResponse pr = new PingResponse();
      return pr;
    }

    public CheckStatusResponse CheckStatus(CheckStatus checkStatus) {
      …
    }

    public SendDischargeSummaryResponse SendDischargeSummary(
      SendDischargeSummary sendDischargeSummary) {
      …
    }

    …
  }
}
```

5.5 Configure the service

5.5.1 Configure the security policy

To configure the use of the policy assertion on the server, extend the `Policy` class and add the policy assertion within the constructor. This can be done by adding an internal class to the service implementation class.

```
internal class ServicePolicy : Policy {
  public ServicePolicy() {
    ExampleDSPolicyAssertion interopAssertion =
      new ExampleDSPolicyAssertion();

    this.Assertions.Add(interopAssertion);
  }
}
```

Next use the `Policy` attribute to specify the server is using the policy class:

```
[WebService(Namespace =
  "http://example.com/ns/2007/DischargeSummaryReceiver/v1")]
[Policy(typeof(ServicePolicy))]
[SoapActor("urn:dsr:wse_server")]
public class Service : IDischargeSummaryReceiverBinding {
  …
}
```

5.5.2 Configure the logical name of the service

Use the `SoapActor` attribute on the service to configure the logical name:

```
…
[Policy(typeof(ServicePolicy))]
[SoapActor("urn:dsr:wse_server")]
public class Service : IDischargeSummaryReceiverBinding {
  …
}
```

5.5.3 Configure the location of the certificates

No configuration needs to be done as WSE searches within the *LocalComputer* store for the certificates it needs when doing validation on the server by default. Note that the default location was chosen for the example and can be changed depending on where the security tokens are stored.

5.6 Packaging and deployment

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

5.6.1 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".

5.6.2 Deploy with IIS

5.6.2.1 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 productions 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.

5.6.2.2 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 "SomeWebsite", the URL for the Web service would be `http://<computer name>:<port>/<alias>/Service.svc`.
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.

5.6.2.3 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

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

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

[MSDN2007] Microsoft Development Network, How to: Create a Custom Policy Assertion that Secures SOAP Messages, 2007, http://msdn2.microsoft.com/en-us/library/aa528788.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, version 1.0, 21 December 2006.

[WCF] Microsoft, Windows Communication Foundation (WCF), http://wcf.netfx3.com.

[WSE] Microsoft, Web Services Enhancements (WSE), http://msdn2.microsoft.com/en-us/webservices/aa740663.aspx.

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

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's 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 server 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 server 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 server 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 “Trusted People” 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 server certificate and select “Next”.
6. Select “Next” again the finally select “Finished”, the certificate will then appear within the “Certificates” folder under “Other People”.

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 server certificates

The server 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 message. The following sections describe how to add the certificates that are required by the server in order to secure and validate messages.

The following steps are required to setup the server certificates:

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

C.3.1 Adding the server 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 server certificate and select “Next”.
6. Select “Next” again the finally select “Finished”, the certificate will then appear within the “Certificates” folder under “Other People”.

C.3.1.1 Set the server certificate permissions

For ASP.NET to be able to access the private key of the server 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 server 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”.

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.