



**Example Technical Implementation
of Interoperable Web Services with
TLS**

WCF

Version 1.0 draft — 1 April 2009

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 © 2009, NEHTA.

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

Table of contents

1	Introduction	1
1.1	Background.....	1
1.2	Purpose.....	1
1.3	Scope	1
1.4	Intended audience	1
1.5	Definitions, acronyms, abbreviations.....	2
1.5.1	Terminology	2
1.6	Style Conventions	2
1.7	Overview	3
2	Example service.....	4
2.1	Organisational	4
2.1.1	Testing	4
2.1.2	Sending discharge summaries.....	5
2.1.3	Checking discharge summary status	5
2.2	Informational	5
2.2.1	Discharge summary.....	5
2.2.2	Status	6
2.3	Technical	6
2.3.1	Informational attributes	6
2.3.2	Behavioural attributes.....	6
2.3.3	Non-functional attributes.....	10
3	Overview of WCF	14
3.1	General background	14
3.2	Technical Overview	14
3.3	Requirements.....	14
3.3.1	Client deployment	14
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	16
4.1	Modify the WSDL files.....	16
4.1.1	WSDL containing the service instance information.....	16
4.2	Generate the proxy from the WSDL files.....	16
4.3	Create a client program	17
4.4	Edit the client configuration	18
4.4.1	Create a custom binding.....	18
4.4.2	Create endpoint behaviour.....	19
4.4.3	Create an endpoint	20
4.5	Implement the client application	21
4.5.1	Add the service certificate validation callback	21
4.5.2	Instantiate the proxy object	22
4.5.3	Invoke the Web service methods.....	22
4.5.4	Close the proxy object	23
4.5.5	Handle exceptions and faults	23
5	Web service	25
5.1	Modify the WSDL files.....	25
5.2	Generate the service interface from the WSDL files.....	25
5.3	Create a WCF service project.....	25
5.4	Implement the Web service.....	26
5.4.1	Create the service implementation class	26

5.4.2	Implement the service methods	26
5.4.3	Add the WS-Addressing MessageID header	26
5.4.4	Throw faults	27
5.5	Edit Service.svc	27
5.6	Edit the service configuration	27
5.6.1	Create a custom binding	27
5.6.2	Create a service behaviour	28
5.6.3	Configure the service behaviour	29
5.6.4	Configure the service endpoint	29
5.7	Package and deploy the Web service	30
5.7.1	Deploy with IIS.....	30
Appendix A: References.....		32
Appendix B: Installation.....		33
B.1	Visual Studio 2008 with SP 1	33
B.2	Internet Information Services on Windows XP.....	33
Appendix C: Key management.....		34
C.1	Setting up the MMC for certificate management	34
C.2	Installing certificates for the client application.....	34
C.2.1	Adding the client certificate	34
C.2.2	Adding the CA	35
C.3	Setting up the service certificates	35
C.3.1	Adding the service certificate	35
C.3.2	Adding the CA	35
C.4	Pitfalls.....	36
Appendix D: Debugging.....		37
D.1	Diagnostics configuration	37
D.2	Displaying service metadata (WSDL)	37
D.3	serviceDebug behaviour element	38
D.4	HTTP Debugging Proxy	38
D.5	Debugging service code that uses IIS	38
D.6	SSL Debugging.....	39

Document information

Change history

Version	Date	Comments
1.0 draft	2009-04-01	Draft

This page is intentionally left blank.

1 Introduction

1.1 Background

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

NEHTA has published a number of technical documents to support the use of Web services. These include the *Web Services Profile* [WSP2009].

1.2 Purpose

This document is part of a series of documents that provide examples of how to build Web services and clients conforming to the *Web Services Profile* [WSP2009]. The *Web Services Profile* document [WSP2009] defines 3 profiles: "Web services profile", "WS-Security profile" and "TLS security profile". This document explains a way of building Web services and clients that conform to the "Web services profile" and "TLS security profile" using Microsoft Windows Communication Foundation (WCF) toolkit [WCF].

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

The main purpose of this document is to support the understanding and interpretation of the conformance criteria in the *Web Services Profile* [WSP2009]. However, it can also assist programmers who are learning how to use WCF.

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

1.3 Scope

This document only covers the WCF [WCF] toolkit and Web services secured with TLS [TLS1999].

There is an example technical implementation document for building Web services secured with TLS using the Java Metro toolkit [METRO]. That example technical implementation can interoperate with this implementation, but it will not be discussed in this document.

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

These example technical implementation documents are not an endorsement of these platforms by NEHTA.

1.4 Intended audience

This document is intended for:

- Software developers; and
- System administrators.

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

The reader is also expected to be familiar with the *Web Services Profile* [WSP2008]. The criteria from [WSP2009] are referred to by their criterion number (e.g. "WS 3.1.1.1-1").

1.5 Definitions, acronyms, abbreviations

HTTP	Hypertext Transfer Protocol
HTTPS	Secure HTTP
IDE	Integrated Development Environment
IIS	Internet Information Services
Metro	Open source Web services stack from Sun
MMC	Microsoft Management Console
SDK	Software Development Kit
TLS	Transport Layer Security
SP	Service Pack
SSL	Secure Sockets Layer
WCF	Windows Communication Foundation
WSDL	Web Service Definition Language

1.5.1 Terminology

This document uses the following terms:

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

1.6 Style Conventions

This document uses the following style conventions:

<i>Italics</i>	<ul style="list-style-type: none"> • Document titles • Program names, tool names • File names, directory paths • URLs
----------------	---

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

1.7 Overview

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

Chapter 3 provides a brief overview of WCF.

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

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

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 WCF programs.

2 Example service

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

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

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

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

2.1 Organisational

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

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

- Sending provider: the program that generates the discharge summary and sends it; and
- Receiving provider: the program that receives the discharge summary.

For the purpose of this example the business process for sending a discharge summary involves three activities:

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

2.1.1 Testing

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

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

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

This is a request-response operation at the technical-level to comply with criterion WS 5.1.5.1-1 from the *Web Services Profile* [WSP2009].

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

2.1.2 Sending discharge summaries

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

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

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

At the technical-level, this operation is implemented as a request-response operation to comply with criterion WS 5.1.5.1-1. That is, a response is sent back, but it contains no business-level information.

2.1.3 Checking discharge summary status

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

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

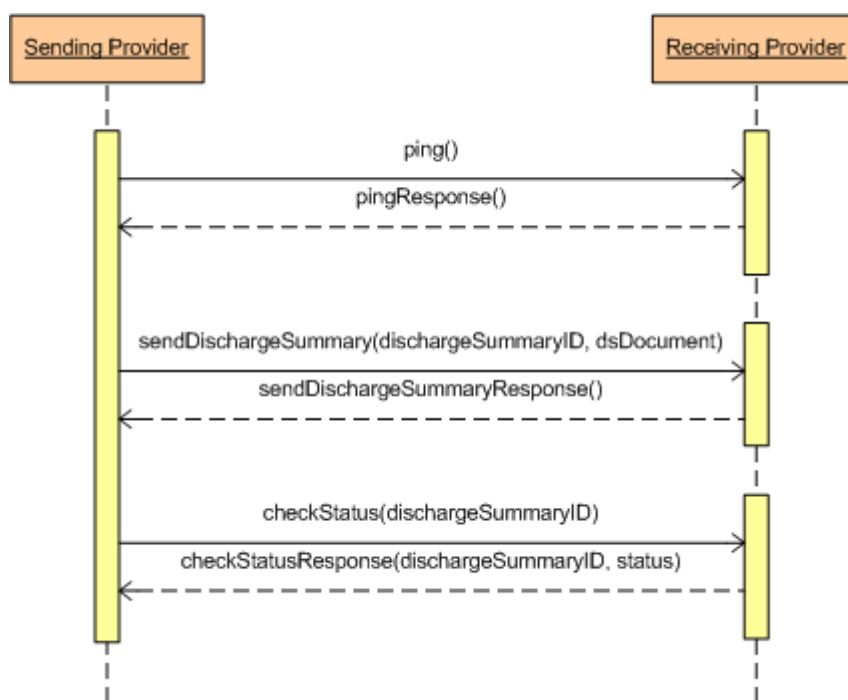


Figure 1: Example discharge summary business workflows

2.2 Informational

2.2.1 Discharge summary

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

The discharge summary data model is simple since the aim of this example is to demonstrate Web services, rather than demonstrate a real discharge summary scenario. NEHTA's *National Discharge Summary Data Content*

Specification [NDS2006] contains much more structured data and metadata in the data model of a discharge summary.

2.2.2 Status

The possible status values for a discharge summary are:

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

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

2.3 Technical

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

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

2.3.1 Informational attributes

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

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

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ns.nehta.gov.au/Example/WSP/DS/Xsd/DischargeSummary/1.0"
  elementFormDefault="qualified">

  <xsd:complexType name="DischargeSummaryType">
    <xsd:sequence>
      <xsd:element name="documentId" type="xsd:string"/>
      <xsd:element name="notes" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

2.3.2 Behavioural attributes

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

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

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

2.3.2.1 WSDL containing service interface information

Criterion WS 3.1.2.1-1 recommends separating the service interface information from the service instance information. This section will go through the WSDL containing the service interface information for our sample Web service.

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

2.3.2.1.1 Header

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

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

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

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

Following criterion WS 3.2.3.1-1, the addressing information in the WSDL is described using *WS-Addressing 1.0 – Metadata* [WSAM2007], whose namespace is:

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

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

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

It is a URL and uses a slash character as the separator, as recommended by criteria WS 3.1.3.1-2 and WS 3.1.3.1-3 respectively. This service namespace is used as the target namespace of the WSDL, following criterion WS 3.1.3.1.4. It is associated with the namespace prefix of `tns` so that it can be referenced in the document. The prefix of the target namespace does not necessarily have to be `tns`; it is just a commonly used convention.

2.3.2.1.2 Types

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

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

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

The WSDL's types section contains a schema defined using the W3C XML Schema language [XSD2004]. The target namespace of this schema is the service's namespace. Since the wrapper elements for the service's operations are declared in this schema, the wrapper elements will belong to the service's namespace, as required by criterion WS 3.1.3.1-8.

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

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

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

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

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

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

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

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

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

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

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

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:sequence>
  </xsd:complexType>
```

```

        <xsd:element name="documentId" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
</wsdl:types>

```

2.3.2.1.3 Messages

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

The messages follow the wrapped convention to conform to criterion WS 5.1.4.1-1. The messages only have one part, where each part references an XML Schema element that was declared in the types section of the WSDL.

```

<wsdl:message name="pingInMsg">
  <wsdl:part name="body" element="tns:ping"/>
</wsdl:message>

<wsdl:message name="pingOutMsg">
  <wsdl:part name="body" element="tns:pingResponse"/>
</wsdl:message>

<wsdl:message name="sendDischargeSummaryInMsg">
  <wsdl:part name="body" element="tns:sendDischargeSummary"/>
</wsdl:message>

<wsdl:message name="sendDischargeSummaryOutMsg">
  <wsdl:part name="body" element="tns:sendDischargeSummaryResponse"/>
</wsdl:message>

<wsdl:message name="checkStatusInMsg">
  <wsdl:part name="body" element="tns:checkStatus"/>
</wsdl:message>

<wsdl:message name="checkStatusOutMsg">
  <wsdl:part name="body" element="tns:checkStatusResponse"/>
</wsdl:message>

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

```

2.3.2.1.4 Port Type

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

Following criterion WS 7.1.2.1-1, the input, output and fault messages of all operations are assigned WS-Addressing Action values. The WS-Addressing Action attributes belong to the namespace of *WS-Addressing 1.0 - Metadata* [WSAM2007]. The values used for the WS-Addressing Action conform to the scheme set out in the following criteria: WS 3.1.3.1-5, WS 3.1.3.1-6 and WS 3.1.3.1-7.

```

<wsdl:portType name="DischargeSummaryReceiver">

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

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

```

```

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

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

</wsdl:portType>

</wsdl:definitions>

```

2.3.3 Non-functional attributes

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

2.3.3.1 WSDL containing service instance information

The non-functional attributes of a service are placed in a separate file as recommended by criterion WS 3.1.2.1-1. This second WSDL file contains the service instance information. It specifies the concrete aspects of the service interface, such as how data is transported and how it is secured.

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

2.3.3.1.1 Header

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

```

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

```

Criterion WS 5.1.1.1-1 recommends the use of SOAP 1.2 as the messaging protocol. This is specified in the WSDL by using the XML namespace for the SOAP 1.2 binding, which is namely:

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

Following criterion WS 3.2.1.1-1, the WSDL uses the *WS-Policy 1.5 Framework* [WSPL2007] to define the non-functional attributes of the service that can be specified formally within the WSDL file. The *wsp* prefix is used to refer to the namespace of *WS-Policy 1.5*, namely:

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

The security policies of the service are specified using *WS-SecurityPolicy 1.2* [WSSPL2007], which follows criterion WS 3.2.2.1-1. The *sp* prefix is used to refer to the namespace of *WS-SecurityPolicy 1.2*, namely:

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

The addressing policies of the service are specified using *WS Addressing 1.0 - Metadata* [WSAM2007], which follows criterion WS 3.2.3.1-1. The `wsam` prefix is used to refer to the namespace of *WS-Addressing 1.0 - Metadata*, namely:

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

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

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

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

2.3.3.1.2 Addressing Policy

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

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

The `Addressing` assertion element indicates the use of *WS-Addressing* [WSAM2007], as per criterion WS 7.1.1.1-1.

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

2.3.3.1.3 Security Policy

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

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

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

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

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

The `TransportToken` assertion element within the `TransportBinding` assertion specifies the policy constraints for the security tokens used in transport-level security. The `HttpsToken` assertion specifies a requirement for security tokens that support the use of HTTPS. HTTPS refers to the use of the HTTP (Hypertext Transfer Protocol) over TLS (Transport Level Security) or SSL (Secure Sockets Layer). The `RequireClientCertificate` attribute specifies that the client must provide a certificate when establishing a TLS session. This matches with criteria WS 8.2.2.2-1 and WS 8.2.2.2-2, which requires the Web service to use a mutually authenticated connection.

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

2.3.3.1.4 Import

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

```
<wsdl:import location="DischargeSummaryReceiverInterface.wsdl"
```

```
namespace="http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0"/>
```

2.3.3.1.5 Binding

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

The addressing and security policies that were defined in sections 2.3.3.1.2 and 2.3.3.1.3 have to be applied to the service. This is done using `wsp:PolicyReference` elements. The `wsp:PolicyReference` elements are applied to the WSDL binding element as recommended by criterion WS 3.2.1.1-2.

Criterion WS 4.1.1.1-1 recommends the use of HTTP 1.1 as the transport protocol. Although the particular HTTP version cannot be specified in the WSDL, the use of HTTP as the transport protocol can be specified by setting the `transport` attribute of the SOAP binding element to:

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

To comply with criterion WS 5.1.2.1-1, the *document/literal* style is used. This is done by setting the `style` attributes of SOAP operation elements to `document` and the `use` attributes of SOAP body and fault elements to `literal`.

To comply with criterion WS 5.1.3.1-1, `soapAction` values are not assigned to any operation. The `soapActionRequired` attributes are set to `false` to indicate that the service does not need `soapAction` values in the requests.

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

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

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

  <wsdl:operation name="Ping">
    <soap:operation style="document" soapActionRequired="false"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>

  <wsdl:operation name="SendDischargeSummary">
    <soap:operation style="document" soapActionRequired="false"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
    <wsdl:fault name="invalidIdFault">
      <soap:fault name="invalidIdFault" use="literal"/>
    </wsdl:fault>
  </wsdl:operation>

  <wsdl:operation name="CheckStatus">
    <soap:operation style="document" soapActionRequired="false"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
    <wsdl:fault name="invalidIdFault">
      <soap:fault name="invalidIdFault" use="literal"/>
    </wsdl:fault>
  </wsdl:operation>

</wsdl:binding>
```

2.3.3.1.6 *Service*

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

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

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

</wsdl:definitions>
```

3 Overview of WCF

3.1 General background

Windows Communication Foundation (WCF) [WCF] is a programming framework for Microsoft .NET that is used for building service-oriented applications that intercommunicate.

3.2 Technical Overview

WCF provides a single service-oriented programming model to unify a number of different distributed communication technologies (such as Web services, .NET Remoting and Microsoft Message Queuing). Some of the benefits of the unification include shorter development time, more concise code, and adherence to more modern standards.

In the WCF model, a service exposes one or more endpoints for accessing the service. An endpoint is defined by an address, a binding and a contract. The address specifies the location of the endpoint, e.g. a network address. The binding specifies how a client can communicate with the service, such as what transport protocol to use and how messages should be encoded. The contract defines the operations that a client can invoke.

WCF includes a utility for generating classes from WSDL files. The utility creates classes for the client that can be used for invoking methods on the service. It also can be used to generate a service interface, which can be used as the basis of a service implementation.

3.3 Requirements

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

3.3.1 Client deployment

- Microsoft .NET Framework 3.5 SP 1.

3.3.2 Web service deployment

The same software is required as the client deployment and;

- Internet Information Services (IIS).

3.3.3 Client development

The same software is required as the client deployment and;

- Microsoft Visual Studio 2008 with SP 1.

3.3.4 Web service development

The same software is required as the Web service deployment and client development.

3.4 Platform used

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

- Microsoft Windows XP SP3

- Microsoft Visual Studio 2008 with SP 1
- Microsoft .NET 3.5 Framework SP 1
- Internet Information Services 5.0

and

- Microsoft Windows Server 2003 SP 1
- Microsoft .NET 3.5 Framework SP1
- Internet Information Services 6.0

4 Web service client

This chapter describes how to build a Web service client that conforms to the *Web Services Profile* [WSP2009] using WCF. The aim of a Web service client is to invoke an operation on a service instance.

The process of creating a Web service client simply involves generating the service interfaces classes from the WSDL files and then using those classes in the client program.

The steps for building a Web service client are:

1. Modify the WSDL files;
2. Generate the proxy from the WSDL files;
3. Create a client program that implements the WCF proxies;
4. Edit the client configuration; and
5. Implement the client application.

4.1 Modify the WSDL files

4.1.1 WSDL containing the service instance information

4.1.1.1 WS-Policy

WCF does not support WS-Policy 1.5 [WSPL2007], so all policy details can be removed from the WSDL containing the service instance information. When using a WSDL that contains policy details that are not supported with the proxy generation utility, warnings will be output stating that the policy details have been ignored, but the code will still be generated.

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

4.2 Generate the proxy from the WSDL files

The proxy is used within a client program to invoke operations on a remote Web service. 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. This means SOAP messages do not have to be created and processed manually.

To generate a WCF proxy from the WSDL files, use the *ServiceModel Metadata Utility Tool*, *svcutil*, which is included with the Windows SDK. The tool can be

found in the `<installation directory>\Microsoft SDKs\Windows\v6.0A\Bin` directory.

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

```
svcutil DischargeSummaryReceiver.wsdl
       DischargeSummaryReceiverInterface.wsdl
       DischargeSummary.xsd
       /serializer:auto
       /namespace:*,Nehta.Example.DS.Client
       /out:DischargeSummaryReceiver.cs
       /config:app.config
```

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 named parameters are:

- `/serializer:` determines how the types will be serialised. The value of `auto` lets `svcutil` determine which serialisation mechanism to use based on information in the WSDL.
- `/namespace:` specifies what namespace the generated source file will be in. The generated class will be enclosed within the specified namespace.
- `/out:` specifies the name and location of the generated source file. The source file contains the service interface specification and the proxy that the client will use to send and receive messages from the service. This file should be added to the client project.
- `/config:` specifies the name and location of the generated configuration file.

The `svcutil` tool generates two files:

- A single C# source code file containing the service interface and proxy code. The proxy code includes the data type and fault classes that are specified within the WSDL. The generated source file can be added to the client project without any modifications.
- A configuration file. The `svcutil` tool generates a default configuration file containing many default values for WCF web services. These configuration files by themselves are of limited use and are usually merged into existing application configuration files.

When the generated proxy class is instantiated within the client application to invoke operations on the Web service, it reads the WCF configuration stored within the application configuration file `app.config`. Although `svcutil` creates a basic configuration file when no WS-Policy information is in the WSDL, that file will not be used. In this example a more sophisticated configuration file will be manually created and edited using the *Microsoft Service Configuration Editor* (as described in section 4.4).

The client can also be configured programmatically but this approach has not been used for this example.

4.3 Create a client program

In the Visual Studio IDE, create a project for the Web service client. There are a number of different ways a client can be implemented (e.g. as a command-line application or a Windows Forms application). It doesn't matter which is used, because this example is only concerned with the Web services part of the application.

In the new project, add references to the following assemblies:

- `System.ServiceModel:` contains the core WCF classes for implementing a Web service and client.

- `System.Runtime.Serialization`: contains classes used for serialising and deserialising types that are used by the Web service and client.

Add the generated source file (that was created in Section 4.1) to the project using the “Add → Existing Item...” function.

Create a new “Application Configuration File” for the project by using the “Add → New Item...” function. This step will create a default configuration file called *App.config* that contains no WCF configuration information. This empty *App.config* file will be set up in section 4.4 using the *Microsoft Service Configuration Editor*.

4.4 Edit the client configuration

The *App.config* file needs to be edited to configure the Web service client. Since it is an XML file, any XML editor or text editor can be used to edit it. However, the easiest way to edit the *App.config* file is using the graphical *Microsoft Service Configuration Editor* tool. This document will describe configuring WCF using this tool. The resulting configuration XML will also be shown.

4.4.1 Create a custom binding

A binding specifies how the client communicates with the Web service. Namely, what transport protocol to use, how messages should be encoded and what protocols and security mechanisms should be used.

WCF provides a number of built-in bindings. However, none of those bindings are suitable therefore, a custom binding is needed.

4.4.1.1 Create the binding

To create a custom binding:

1. Right-click on the *App.config* file in Visual Studio and then select the “Edit WCF Configuration...” menu item. This opens the file in the *Microsoft Service Configuration Editor*.
2. Right-click on the “Bindings” folder in the “Configuration” tree pane and select the “New Binding Configuration...” menu item.
3. In the “Create a New Binding” dialog box that appears, select “customBinding” and click the “OK” button.
4. Enter a name for the new binding in the “Name” field. (In this example, the arbitrary name of “TlsBinding” is used.)
5. The newly-created binding will automatically have two binding element extensions. The “textMessageEncoding” extension specifies how messages are encoded. By default, it’s set to use SOAP 1.2 and WS-Addressing 1.0, which is aligned with criteria WS 5.1.1.1-1 and WS 7.1.1.1-1. The “httpTransport” extension specifies that HTTP is to be used as the transport protocol.

4.4.1.2 Configure security

To add and configure security for the custom binding:

1. Left-click on the new custom binding (in this example, “SecureBinding”) in the “Configuration” tree pane.
2. Select the “httpTransport” element in the “Binding element extension position” section and click “Remove”. This will remove the HTTP transport binding element extension.
3. In the “Binding element extension position” section, click the “Add...” button.

4. In the “Adding Binding Element Extension Sections” dialog box that appears, select “httpsTransport” and click the “Add” button. This aligns with criteria WS 8.2.1.1-1.
5. Double-click on the “httpsTransport” element in the “Binding element extension position” to edit the configuration.
6. Change the “RequireClientCertificate” property to “true”. This enables mutual authentication which aligns with criteria WS 8.2.2.1-1.

4.4.1.3 Save changes

Save the changes that were made to *App.config* file by the *Microsoft Service Configuration Editor*.

The following XML fragment will be added to `bindings` element within the `system.serviceModel` element in the *App.config* file:

```
<customBinding>
  <binding name="TlsBinding">
    <textMessageEncoding />
    <httpsTransport requireClientCertificate="true" />
  </binding>
</customBinding>
```

4.4.2 Create endpoint behaviour

4.4.2.1 Create the behaviour

To create an endpoint behaviour:

1. Right-click on the *App.config* file in Visual Studio and then select the “Edit WCF Configuration...” menu item. This opens the file in the *Microsoft Service Configuration Editor*.
2. Expand the “Advanced” folder in the “Configuration” tree pane.
3. Right-click on the “Endpoint Behaviors” folder in the “Configuration” tree pane and select the “New Endpoint Behavior Configuration...” menu item.
4. Enter a name for the new endpoint behaviour in the “Name” field. (In this example, the arbitrary name of “TlsBehavior” is used.)

4.4.2.2 Set the client certificate

WCF uses the Windows certificate store to access the keys and certificates. The required security tokens (i.e. the key and certificate) must be installed into the Windows certificate store before the application can be run. Security token installation is covered in Appendix C: Key Management.

The Web service client determines which key and certificate to use from the values configured in the endpoint behaviour.

To set the client security token in the endpoint behaviour:

1. Click on the new endpoint behaviour (in this example, it is `TlsBehavior`) in the “Configuration” tree pane.
2. In the “Behavior element extension position” section, click the “Add...” button.
3. In the “Adding Behavior Element Extension Sections” dialog box that appears, select “clientCredentials” and click the “Add” button.
4. Double-click on the “clientCredentials” element in the “Behavior element extension position”.
5. Expand the “clientCredentials” element in the “Configuration” tree pane.
6. Select the “clientCertificate” element in the “Configuration” tree pane.

7. Leave the "StoreLocation" property value set to "CurrentUser". This value refers to the current user's certificate store, which is meant to store client certificates.
8. Leave the "StoreName" property value to set to "My". This value refers to the "Personal" folder in the certificate store.
9. Leave the "X509FindType" property value set to "FindByThumbprint". The Windows certificate store will be searched using the thumbprint of the X.509 certificate.
10. Set the "FindValue" property to the thumbprint of the client certificate. In this example the value is:
"0f bc f1 a9 31 af 65 59 4e 56 85 a6 85 27 b2 3e 51 80 2b 3d". There must be a private key associated with this certificate in the certificate store.

4.4.2.3 Save changes

Save the changes that were made to the *App.config* file. The following XML fragment will be added to the `behaviors` element within the `service.serviceModel` element in the *App.config* file:

```
<behaviors>
  <endpointBehaviors>
    <behavior name="TlsBehavior">
      <clientCredentials>
        <clientCertificate
          findValue="0f bc f1 a9 31 af 65 59 4e 56 85 a6 85 27 b2 3e 51 80 2b 3d"
          storeLocation="CurrentUser" storeName="My"
          x509FindType="FindByThumbprint" />
        </clientCredentials>
      </behavior>
    </endpointBehaviors>
  </behaviors>
```

4.4.3 Create an endpoint

To access a Web service, an endpoint for the Web service must be created in the client configuration. An endpoint must have a binding, which specifies how to communicate with the Web service. Section 4.4.1 explained how to create a custom binding that conforms to *Web Services Profile* [WSP2009] "Tls Profile". The created endpoint will have to be associated with this binding.

An endpoint must also have a contract, which specifies what operations the client can call. The created endpoint will have to be associated with the service interface, which is in the source file generated in Section 4.1.

An endpoint must have an address, which specifies the location where the Web service can be accessed. This address can be a logical or physical address. This example uses the physical address of the Web service.

An endpoint can also have an endpoint behaviour, which allows modification of the client run-time behaviour. The created endpoint will have to be associated with the behaviour created in Section 4.4.2. This behaviour specifies what security tokens are to be used when using TLS.

To create the endpoint:

1. Right-click on the *App.config* file in Visual Studio and then select the "Edit WCF Configuration..." menu item. This opens the file in the *Microsoft Service Configuration Editor*.
2. Expand the "Clients" folder in the "Configuration" tree pane.
3. Right-click on the "Endpoints" folder in the "Configuration" tree pane and select the "New Client Endpoint" menu item.
4. Enter a name for the new endpoint in the "Name" property. (In this example, it is `ServiceEndpoint`.)

5. Enter the service's physical address (in this example, it is `https://localhost/TlsService/Service.svc`) in the "Address" property.
6. Select the endpoint behaviour that was created in Section 4.4.2 (in this example, it is `TlsBehavior`) in the "BehaviorConfiguration" property.
7. Select "customBinding" in the "Binding" property and select the binding that was created in Section 4.4.1 (in this example, it is `TlsBinding`) in the "BindingConfiguration" property.
8. Enter the full type name of the service interface for the Web service in the "Contract" property. This can be found by looking in the generated source file that was added to the project. In the example, the full type name of the Web service interface is:
`Nehta.Example.DS.Client.DischargeSummaryReceiver`.

4.4.3.1 Save changes

Save the changes that were made to the *App.config* file. The following XML fragment will be added to the `client` element within the `service.serviceModel` element in the *App.config* file:

```
<endpoint address="https://localhost/TlsService/Service.svc"
  behaviorConfiguration="TlsBehavior"
  binding="customBinding" bindingConfiguration="TlsBinding"
  contract="Nehta.Example.DS.Client.DischargeSummaryReceiver"
  name="ServiceEndpoint" />
```

4.5 Implement the client application

To invoke an operation on the service, the client needs to:

1. Add the service certificate validation callback;
2. Instantiate the proxy object;
3. Invoke the Web service methods; and
4. Close the proxy object.

4.5.1 Add the service certificate validation callback

The service certificate validation callback is used to validate a service certificate when invoking a method on a service that uses TLS. The callback is invoked during the TLS negotiation by the .NET framework.

The callback has a number of parameters that can be used to determine if the certificate is valid:

- `object sender`: The request details of the service being invoked.
- `X509Certificate2 certificate`: The service certificate.
- `X509Chain chain`: The certificate chain of the service certificate.
- `SslPolicyErrors sslPolicyErrors`: The service certificate policy errors enumeration. Note the enumeration allows bitwise combination of errors.
 - `SslPolicyErrors.None`: This indicates there were no problems with the service certificate.
 - `RemoteCertificateNameMismatch`: This indicates the service certificate subject common name (CN) does not match the host name of the service.
 - `RemoteCertificateChainErrors`: This indicates there was an error when building the chain for the service certificate.

- `RemoteCertificateNotAvailable`: This indicates the service certificate is not available.

The callback has a `bool` return type that is used to indicate the validity of the service certificate. When `true` is returned it indicates that the service certificate is valid and the connection should continue. When `false` is returned it indicates the service certificate is not valid and the connection should be aborted.

The following code fragment is the certificate validation callback from the example. It first checks if there were any policy errors and returns `true` if there are none. It then checks if there was a `RemoteCertificateNameMismatch` error and if there was, the sender is checked to see if it's from `localhost`. When it is, `true` is returned. All other errors will return `false`.

```
public bool ValidateServiceCertificate(
    object sender, X509Certificate certificate, X509Chain chain,
    SslPolicyErrors sslPolicyErrors)
{
    // Check if there were any errors
    if (sslPolicyErrors == SslPolicyErrors.None)
    {
        // Return true as there were no errors
        return true;
    }

    // The 'RemoteCertificateNameMismatch' policy error occurs when
    // the service certificate subject CN value does not equal the
    // host name of the service.
    if (sslPolicyErrors == SslPolicyErrors.RemoteCertificateNameMismatch)
    {
        // Check if the name mismatch policy error is because of 'localhost'
        HttpRequest webRequest = (HttpRequest)sender;
        if (webRequest.Address.Host == "localhost")
        {
            // Connecting to 'localhost' so return true
            return true;
        }
    }

    // Return false meaning the service certificate is invalid and the connection
    // should be aborted
    return false;
}
```

The callback is set on the `ServerCertificateValidationCallback` property of the `ServicePointManager` static class. This can be done using the `RemoteCertificateValidationCallback` delegate. The callback must be set before a method on the proxy is invoked. Once the TLS connection has been validated and established it is cached and re-used later for subsequent proxy invocations.

```
ServicePointManager.ServerCertificateValidationCallback +=
    new RemoteCertificateValidationCallback(ValidateServiceCertificate);
```

4.5.2 Instantiate the proxy object

The proxy is instantiated using `new`, just like any normal object. This is shown in the code fragment below.

The proxy will load the WCF configuration from the `App.config` file.

```
DischargeSummaryReceiverClient dsrc = new DischargeSummaryReceiverClient();
```

4.5.3 Invoke the Web service methods

The client invokes the Web service operations by calling the corresponding methods on the proxy object.

```
// Invoke "ping"
```

```

try
{
    dsrc.ping(new ping());
}
catch (...)
{
    ...
}

// Invoke "sendDischargeSummary"

// Create the discharge summary document itself
DischargeSummaryType ds = new DischargeSummaryType();
ds.notes = "This is an example discharge summary";
ds.documentId = "someid"; // a unique ID for the discharge summary

// Create and set parameters
sendDischargeSummary sds = new sendDischargeSummary();
sds.document = ds;

// Invoke the operation using the client stub "dsrc"
try
{
    dsrc.sendDischargeSummary(sds);
}
catch (...)
{
}

// Invoke "checkStatus"

// Create and set parameters
checkStatus cs = new checkStatus();
cs.documentId = "someid";

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

```

4.5.4 Close the proxy object

When the proxy object is no longer needed, it should be closed. This frees any resources being used by the proxy. A "using" block can also be used to ensure the proxy is closed.

```
dsrc.Close();
```

4.5.5 Handle exceptions and faults

Within the WSDL, each operation can specify what SOAP faults can occur when calling that particular operation. When a client invokes a Web service method and a fault occurs, the corresponding exception can be caught using the `FaultException` class and a standard C# try/catch block. The service utility generates a fault class for each fault defined within the WSDL. The classes generated do not inherit from the standard `Exception` class.

To catch a specific fault that's defined within the WSDL:

```

try
{
    ...
}
catch (FaultException<InvalidIdFault> ex)
{
    // Handle the error
}

```

To catch all faults that are defined within the WSDL:

```

try
{

```

```
...
}
catch (FaultException ex)
{
    // Handle the error
}
```

The WCF documentation also recommends catching the `TimeoutException` and `CommunicationException` exceptions when invoking operations or calling `Close`. These must be handled after the specific faults. It also recommends calling `Abort` within any catch block to ensure that proper shutdown of the proxy can take place when a failure has occurred.

The typical pattern for handling exceptions is given below:

```
// Create the proxy

try {
    // Calls on the proxy methods
    ...

    // Close the proxy
    proxy.Close()
}
catch (FaultException<InvalidIdFault> ex)
{
    // Catch any specific exception
    ...
    proxy.Abort();
}
catch (FaultException ex)
{
    // Catch all WSDL defined faults
    ...
    proxy.Abort();
}
catch (TimeoutException ex)
{
    ...
    proxy.Abort();
}
catch(CommunicationException ex)
{
    ...
    proxy.Abort();
}
```

5 Web service

This section describes how to build a WCF Web service that conforms to the *Web Services Profile* [WSP2009].

A WCF Web service can be implemented with the following steps:

1. Modify the WSDL files;
2. Generate the service interface from the WSDL files;
3. Create a WCF service project;
4. Implement the Web service;
5. Edit *Service.svc*;
6. Edit the service configuration; and
7. Package and deploy the Web service.

5.1 Modify the WSDL files

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

5.2 Generate the service interface from the WSDL files

The interface declaration for the service can be generated from the WSDL files using the *svcutil* command line tool.

Section 4.1 describes how to use this tool on the client-side to generate the service interface and proxy code. The steps described in Section 4.1 are similar for the Web service. The only difference is that a different namespace is used for the generated code. The namespace for the service is `Nehta.Example.DS.Service`. However, it is also possible to use the same generated code for the both the Web service and client with the same namespace.

Like the client-side, the generated configuration file is ignored. Only the generated source file will be used.

5.3 Create a WCF service project

Create a new Web Site project in the Visual Studio IDE. Ensure that the "WCF Service" template is selected.

There are a number of ways to host a WCF service. The descriptions in this document are in terms of an IIS-hosted WCF service, which is created by selecting the default "HTTP" option in the "Location" field. The project's Solution name should be typed in after the "http://localhost/" string in the "HTTP" location value.

A WCF service can be written in different programming languages. The code examples in this document are in C#, which is the default option for the "Language" field.

Once the new project is created, add the source file that was generated in Section 5.2 to it. Unlike the client, there is no need to add the `System.ServiceModel` and `System.Runtime.Serialization` assemblies since this is automatically done by selecting the "WCF Service" template.

5.4 Implement the Web service

When Visual Studio creates the initial WCF service project, it creates a default service that can be modified. The new WCF service project will come with a *Service.cs* file in the *App_Code* folder that contains a sample Web service implementation. Apart from the `using` statements at the top, the contents of this file can be deleted. This section will explain how to then build this service implementation class.

5.4.1 Create the service implementation class

The example source code below shows the class declaration of the service implementation. It should implement the service interface, which is found in the source file generated in Section 5.1. It should have a "ServiceBehavior" attribute, which specifies the WSDL's target namespace.

```
namespace Nehta.Example.DS.Service {
    [ServiceBehavior(
        Namespace = "http://ns.nehta.gov.au/Example/WSP/DS/Svc/Receiver/1.0")
        InstanceContextMode = InstanceContextMode.Single,
        ConcurrencyMode = ConcurrencyMode.Multiple)]
    public class DSRService : DischargeSummaryReceiver {
        ...
    }
}
```

5.4.2 Implement the service methods

Implementations need to be provided for all the methods in the service interface.

```
public pingResponse ping(pingRequest request)
{
    ServiceUtil.SetMessageIdHeader();
    ... // Process request and return response

    pingResponse result = new pingResponse();
    return result;
}

public sendDischargeSummaryResponse sendDischargeSummary(
    sendDischargeSummaryRequest request)
{
    ServiceUtil.SetMessageIdHeader();
    ... // Process request and return response
}

public checkStatusResponse checkStatus(checkStatusRequest request)
{
    ServiceUtil.SetMessageIdHeader();
    ... // Process request and return response
}
```

5.4.3 Add the WS-Addressing MessageID header

Since WCF by default doesn't add the WS-Addressing `MessageID` header to SOAP responses, it needs to be explicitly set in the current operation context, so as to satisfy criterion WS 7.1.3.1-1 and WS 7.1.3.1-2. Since this code will have to be called in each Web service method implementation, in this example it is placed in a separate method with a meaningful name. The identifier used must be globally unique.

```
private void SetMessageIdHeader() {
    OperationContext.Current.OutgoingMessageHeaders.MessageId = new UniqueId();
}
```

5.4.4 Throw faults

When the *svcutil* tool generates the service interface, it generates classes for any faults that are specified in the WSDL. These classes are only used to specify the details of the fault and are not thrown directly.

To use the fault in the service implementation, create an instance of the generated class and set the details of the fault. Create a `FaultException` object using the created fault object and specify the reason for the fault by creating a `FaultReason` object. The fault exception can then be thrown using the normal `throw` statement.

```
public static void ThrowInvalidIdFault(string id) {
    // Set the data associated with the fault
    InvalidIdFault iif = new InvalidIdFault();
    iif.FaultDescription = InvalidIdFaultMessage;
    iif.DocumentId = id;

    // Create and throw the fault
    FaultException fe =
        new FaultException<InvalidIdFault>(iif,
            new FaultReason("Reason: " + InvalidIdFaultMessage));

    throw fe;
}
```

5.5 Edit Service.svc

With Visual Studio, another file that gets generated by default when creating a WCF Service project is the *Service.svc* file. This defines the file that contains the service class and the class itself.

Open *Service.svc* and put the full type name in the `Service` attribute. The full type name includes the namespace and the service type name. Ensure that the `CodeBehind` attribute has the correct name of the source file that contains the service implementation class. By default this is set to *Service.cs* and should only need to be changed if the service source filename has been changed.

The contents of the *Service.svc* file for the example Web service is given below.

```
<% @ServiceHost Language=C# Debug="true"
Service="Nehta.Example.DS.Server.DSRService" CodeBehind="~/App_Code/Service.cs" %>
```

5.6 Edit the service configuration

The new project will have a *Web.config* file, which stores the configuration for the Web service. Since it is an XML file, it can be edited by XML editors or plain text editors. However, the *Microsoft Service Configuration Editor* provides the easiest way to edit this file.

To configure the Web service:

1. Create a custom binding;
2. Create a service behaviour;
3. Configure the service behaviour; and
4. Configure the endpoint.

5.6.1 Create a custom binding

A WCF binding specifies how the client and Web service should communicate with each other, namely what transport protocol to use, how messages should be encoded and what protocols, e.g. security, should be used. Thus, the bindings in the client and service configurations should match each other.

Section 4.4.1 describes how to create a custom binding that adheres to *Web Services Profile* [WSP2009] on the client-side. Follow the steps in Section 4.4.1 for the server-side as well. The only difference is that the file to be edited is *Web.config*, instead of *App.config*.

5.6.2 Create a service behaviour

5.6.2.1 Create the behaviour

To create the service behaviour:

1. Right-click on the *Web.config* file in Visual Studio and select the "Edit WCF Configuration..." menu item. This opens the file in the *Microsoft Service Configuration Editor*.
2. Expand the "Advanced" folder in the "Configuration" tree pane.
3. Right-click on the "Service Behaviors" folder in the "Configuration" tree pane and select the "New Service Behavior Configuration..." menu item.
4. Enter a name for the new service behaviour in the "Name" field. (In this example, it is `TlsBehavior`).

5.6.2.2 Publish service metadata

By default, WCF doesn't make the WSDL accessible.

To publish the service's WSDL:

5. Click on the new service behaviour (in this example, it is `TlsBehavior`) in the "Configuration" tree pane.
6. In the "Behavior element extension position" section, click the "Add..." button.
7. In the "Adding Behavior Element Extension Sections" dialog box that appears, select "serviceMetadata" and click the "Add" button.
8. Expand the "`TlsBehavior`" and then the "serviceMetadata" elements in the "Configuration" tree pane.
9. Select the "serviceMetadata" element in the "Configuration" tree pane.
10. Set the "HttpsGetEnabled" property to true.

5.6.2.3 Add debug details

The debug behaviour element extension is useful when debugging a WCF service. It allows exception details to be sent to client when faults occur which aid in debugging.

To add the debug behaviour element extension:

1. Click on the new service behaviour (in this example, it is `TlsBehavior`) in the "Configuration" tree pane.
2. In the "Behavior element extension position" section, click the "Add..." button.
3. In the "Adding Behavior Element Extension Sections" dialog box that appears, select "serviceDebug" and click the "Add" button.
4. Set the "IncludeExceptionDetailInFaults" to true.

5.6.2.4 Configure security credentials

The service credentials do not need to be added to the configuration of the service as these are handled by IIS. Configuring IIS to support SSL is covered in section 5.7.1.3.

5.6.2.5 Save changes

Save the changes that were made to *Web.config* file. The following XML fragment will be added to `behaviors` element within the `service.serviceModel` element in the *Web.config* file:

```
<serviceBehaviors>
  <behavior name="TlsBehavior">
    <serviceDebug includeExceptionDetailInFaults="true" />
    <serviceMetadata httpsGetEnabled="true" />
  </behavior>
</serviceBehaviors>
```

5.6.3 Configure the service behaviour

To configure the service:

1. Right-click on the *Web.config* file in Visual Studio and select the "Edit WCF Configuration..." menu item. This opens the file in the *Microsoft Service Configuration Editor*.
2. Expand the "Services" folder in the "Configuration" tree pane. By default, WCF creates a "MyService" service.
3. Select the "MyService" element in the "Configuration" tree pane.
4. Enter the full type name of the service implementation class in the "Name" property. (In this example, it is `Nehta.Example.DS.Service.DSRService`).
5. Select the service behaviour that was created in Section 4.3.2 (in this example, it is `TlsBehavior`) in the "BehaviorConfiguration" property.

5.6.3.1 Save Changes

Save the changes that were made to the *Web.config* file.

```
<service behaviorConfiguration="TlsBehavior"
  name="Nehta.Example.DS.Service.DSRService">
  ...
</service>
```

5.6.4 Configure the service endpoint

A Web service must declare an endpoint in order for clients to access it. An endpoint is made up of an address, which represents the location to send SOAP messages, a binding, which specifies how to communicate with the service, and a contract, which specifies the available operations.

To configure the service endpoint:

1. Expand the service element (in this example, it is `Nehta.Example.DS.Service.DSRService`) and then the "Endpoints" element in the "Configuration" tree pane.
2. Select the existing endpoint that was created by default; it has the label "(Empty Name)".
3. Enter a name for the endpoint in the "Name" property. (In this example, it is `ServiceEndpoint`).
4. Select "customBinding" in the "Binding" property and select the binding that was created in section 5.6.1 (in this example, it is `TlsBinding`) in the "BindingConfiguration" property.
5. Enter the full type name of the service interface in the "Contract" property. In the example, the full type name of the Web service interface is `Nehta.Example.DS.Service.DischargeSummaryReceiver`.

5.6.4.1 Save changes

Save the changes that were made to *Web.config* file.

```
<endpoint address="" binding="customBinding" bindingConfiguration="TlsBinding"
contract="Nehta.Example.DS.Service.DischargeSummaryReceiver" />
```

5.7 Package and deploy the Web service

Once the Web service has been created and configured, it can be deployed on Internet Information Services (IIS). The Visual Studio development server cannot be used as it does not support HTTPS.

5.7.1 Deploy with IIS

5.7.1.1 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 binaries. Generally source files are used for debugging and testing, and the compiled files for production systems.

To create a Web service binary distribution:

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

5.7.1.2 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.

To create a virtual directory using the IIS snap-in:

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 "TlsService" for the Web service. 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.

5.7.1.3 Configuring TLS

TLS is configured using the IIS snap-in for the MMC.

1. Right-click on "Default Web Site" and select "Properties".
2. Select the "Directory Security" tab.
3. Within the "Secure communications" section select "Server Certificate..." and select "Next".
4. Select "Assign an existing certificate" and select "Next". Server certificates must be within the Windows certificate store under "Local Computer" "My". Security token installation is covered in Appendix C.
5. On the "Available certificates" dialog select the "tls_server.example.com" certificate and select "Next" and then "Finish".
6. Now that the server certificate has been assigned, the security configuration must be set. This can be set for every virtual directory or for a single virtual directory. The following steps are for a specific virtual directory.
7. Right-click on the "TlsService" virtual directory and select "Properties".
8. Select the "Directory Security" tab.
9. Select "Edit..." within the "Secure communications" section and the security configuration dialog will appear.
10. Check the "Require secure channel (SSL)" checkbox. This aligns with criteria WS 8.2.1.2-1.
11. Check the "Require 128-bit encryption".
12. Within the "Client Certificates" section select "Require client certificates". This enables mutual authentication. This aligns with criteria WS 8.2.2.2-1 and WS 8.2.2.2-2.
13. Select "Ok" on the dialog and then "Apply" on the "Directory Security" tab.

The configuration can be tested by using a browser to access the URL.

5.7.1.4 Disabling CRL Checking

By default CRL checking of client certificates is enabled within IIS when using TLS. When a client certificate contains a CRL distribution point, IIS will attempt to use the location to check the revocation status of the certificate. When a certificate contains an invalid CRL distribution point, the automatic CRL checking needs to be disabled otherwise the server will return the "403 Forbidden" status code. See the knowledge base article [KB294305] for more information.

To disable CRL checking for IIS refer to the article [CRTCHK] for more information.

Appendix A: References

- [CRTCHK] CertCheckMode (Metabase property),
<http://msdn.microsoft.com/en-us/library/ms525603.aspx>
- [GIIWS2007] NEHTA, *Guidelines for Implementing Interoperable Web Services*, version 1.0, March 2007.
- [JAXWS] Sun Microsystems, *Java API for XML Web Services (JAX-WS)*,
<https://jax-ws.dev.java.net>.
- [KB294305] IIS returns HTTP "403.13 Client Certificate Revoked"...,
<http://support.microsoft.com/kb/294305>
- [MS7331123.0] Microsoft, .Net 3.0, WCF, Data Contract Schema Reference.
[http://msdn.microsoft.com/en-us/library/ms733112\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms733112(VS.85).aspx)
- [MS7331123.5] Microsoft, .Net 3.5, WCF, Data Contract Schema Reference.
<http://msdn.microsoft.com/en-us/library/ms733112.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://netfx3.com>.
- [WSAW2006] W3C, *Web Services Addressing 1.0 –WSDL Binding*, W3C Candidate Recommendation, 29 May 2006,
<http://www.w3.org/TR/2006/CR-ws-addr-wsdl-20060529/>.
- [WSP2009] NEHTA, *Web Services Profile*, version 3.1 draft, 20 February 2009.
- [WSPL2006] XmlSoap, *Web Services Policy Framework*, version 1.2, Draft, March 2006,
<http://specs.xmlsoap.org/ws/2004/09/policy/ws-policy.pdf>.
- [WSPL2007] W3C, *Web Services Policy 1.5 – Framework*, W3C Recommendation, 4 September 2007,
<http://www.w3.org/TR/2007/REC-ws-policy-20070904>.
- [WSSPL2005] XmlSoap, *Web Services Security Policy Language (WS-SecurityPolicy)*, Draft, July 2005,
<http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf>.
- [WSSPL2007] OASIS, *WS-SecurityPolicy 1.2*, OASIS Standard, 1 July 2007,
<http://docs.oasis-open.org/ws-sx/wssecuritypolicy/200702/ws-securitypolicy-1.2-spec-os.pdf>.
- [MS7331123.0] Microsoft, .NET 3.0, WCF, Data Contract Schema Reference.
<http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/>.

Appendix B: Installation

The following sections detail how to install the tested environment. All the software was installed on Microsoft Windows XP SP3.

B.1 Visual Studio 2008 with SP 1

1. Install Visual Studio 2008 from the installation media.
2. Download the Visual Studio 2008 SP 1.

Web Install URL:

<http://www.microsoft.com/downloads/details.aspx?FamilyId=FBEE1648-7106-44A7-9649-6D9F6D58056E&displaylang=en>

ISO URL:

<http://www.microsoft.com/downloads/details.aspx?FamilyId=27673C47-B3B5-4C67-BD99-84E525B5CE61&displaylang=en>

3. Run the service pack to install. This also updates the .NET framework to 3.5 SP 1.

B.2 Internet Information Services on Windows XP

1. Insert the Windows XP 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. 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 certificates 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 MMC for certificate management

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 certificate, and a CA (certificate authority) certificate for verification. The follow sections describe how to add the client certificate.

The following steps are required to setup the client certificate:

1. Add the client certificate (that has a private key).
2. Add the CA.

C.2.1 Adding the client certificate

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

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

C.3 Setting up the service certificates

The Web service requires a certificate for use with TLS.

The following steps are required to setup the service certificates:

1. Add the service certificate (that has a private key).
2. Add the CA.

C.3.1 Adding the service certificate

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

C.3.2 Adding the CA

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

6. Select "Next" again and finally select "Finished", the certificate will then appear within the "Certificates" folder under "Trusted Root Certification Authorities".

C.4 Pitfalls

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

Appendix D: Debugging

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

1. Diagnostics configuration;
2. Displaying service metadata (WSDL);
3. `serviceDebug` behaviour element;
4. HTTP debugging proxy;
5. Debugging service code that uses IIS;
6. SSL debugging.

D.1 Diagnostics configuration

When implementing and testing a client or Web service, message and trace logging can be enabled and configured. The configuration for the diagnostics can be added using the service configuration tool or added directly to the `web.config` file.

To add diagnostics configuration using the service configuration tool:

1. Open the service configuration tool on the service applications configuration file, `web.config`.
2. Left click on the "Diagnostics" folder. In the left panel a number of diagnostics options will be displayed.
3. Left click on the "Enable Log Auto Flush" link that is next to the "Log Auto Flush" text to enable log flushing.
4. Left click on the "Enable Message Logging" option next to the "Message Logging" text to enable message level logging. The "Log file:" text specifies where the file will be created. Use the `svctraceviewer.exe` utility to view the log file.
5. Left click on the "Enable Tracing" option next to the "Tracing" text to enable tracing. The "Log file:" text specifies where the file will be created. Use the `svctraceviewer.exe` utility to view the log file.

D.2 Displaying service metadata (WSDL)

When a service is initially created, by default the WSDL is not accessible. To allow the WSDL for a service to be accessible, the `serviceMetadata` behaviour element extension needs to be added to the behaviour used by the service.

To add the `serviceMetadata` behaviour element extension:

1. Open the service configuration tool on the service applications configuration file, `web.config`.
2. Expand the "Advanced" folder.
3. Expand the "Service Behaviors" folder.
4. Right click on the behaviour that's being used for the service and select "Add Service Behavior Element Extension".
5. From the list select `serviceMetadata` to add it to the behaviour.
6. Left click on the `serviceMetadata` element
7. Set the "HttpGetEnabled" to true. Set the "HttpsGetEnabled" to true if using HTTPS. This allows the WSDL to be accessed via HTTP.

D.3 serviceDebug behaviour element

On the service side, the `serviceDebug` element can be added to the service behaviour to allow a HTML help page to be displayed when the service URL is contacted with a browser and to allow fault details be included when SOAP faults are thrown from the service.

To add the `serviceDebug` behaviour element:

1. Open the service configuration tool on the *web.config*.
2. Expand the "Advanced" folder.
3. Expand the "Service Behavior" folder.
4. Right click on the behaviour that is being used for the service and select "Add Service Behavior Element Extension".
5. From the list select the `serviceDebug` behaviour element extension so it is added to the behaviour.
6. Left click on the `serviceDebug` element that has been added.
7. Set the "IncludeExceptionDetailInFaults" to true and leave the rest as their default values.

D.4 HTTP Debugging Proxy

A HTTP debugging proxy can be used to view all messages that are sent and received to and 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 using the service configuration tool.

To configure the client to use the HTTP debugging proxy:

1. Open the service configuration tool on the *app.config*.
2. Expand the "Binding" folder.
3. Expand the binding that is being used for the client.
4. Left click on the "httpTransport" element to view the associated configuration.
5. Set the "UseDefaultWebProxy" to false. When this is set to true the client tries to use the proxy that has been configured within Internet Explorer.
6. Within the "ProxyAddress" field, enter the address of the proxy that's being used. Specify the computer name (even if using the local machine) and the port of the debugging proxy.

D.5 Debugging service code that uses IIS

To debug a service that's running under IIS:

1. Deploy the service to IIS.
2. From within Visual Studio, select "Debug" 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.

D.6 SSL Debugging

Microsoft offers a toolset for debugging SSL issues when using IIS called "SSL Diagnostics". This can be downloaded from the Microsoft IIS site.