More on WSDL

**WSDL Schema**

<xs:schema xmlns:xs=*"http://www.w3.org/2001/XMLSchema"*

xmlns:wsdl=*"http://schemas.xmlsoap.org/wsdl/"* targetNamespace=*"http://schemas.xmlsoap.org/wsdl/"*

elementFormDefault=*"qualified"*>

<xs:complexType mixed=*"true"* name=*"tDocumentation"*>

<xs:sequence>

<xs:any minOccurs=*"0"* maxOccurs=*"unbounded"* processContents=*"lax"* />

</xs:sequence>

</xs:complexType>

<xs:complexType name=*"tDocumented"*>

<xs:annotation>

<xs:documentation>

This type is extended by component types to allow them to be documented

</xs:documentation>

</xs:annotation>

<xs:sequence>

<xs:element name=*"documentation"* type=*"wsdl:tDocumentation"*

minOccurs=*"0"* />

</xs:sequence>

</xs:complexType>

<xs:complexType name=*"tExtensibleAttributesDocumented"*

abstract=*"true"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tDocumented"*>

<xs:annotation>

<xs:documentation>

This type is extended by component types to allow attributes from

other namespaces to be added.

</xs:documentation>

</xs:annotation>

<xs:anyAttribute namespace=*"##other"*

processContents=*"lax"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tExtensibleDocumented"* abstract=*"true"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tDocumented"*>

<xs:annotation>

<xs:documentation>

This type is extended by component types to allow elements from other

namespaces to be added.

</xs:documentation>

</xs:annotation>

<xs:sequence>

<xs:any namespace=*"##other"* minOccurs=*"0"* maxOccurs=*"unbounded"*

processContents=*"lax"* />

</xs:sequence>

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:element name=*"definitions"* type=*"wsdl:tDefinitions"*>

<xs:key name=*"message"*>

<xs:selector xpath=*"wsdl:message"* />

<xs:field xpath=*"@name"* />

</xs:key>

<xs:key name=*"portType"*>

<xs:selector xpath=*"wsdl:portType"* />

<xs:field xpath=*"@name"* />

</xs:key>

<xs:key name=*"binding"*>

<xs:selector xpath=*"wsdl:binding"* />

<xs:field xpath=*"@name"* />

</xs:key>

<xs:key name=*"service"*>

<xs:selector xpath=*"wsdl:service"* />

<xs:field xpath=*"@name"* />

</xs:key>

<xs:key name=*"import"*>

<xs:selector xpath=*"wsdl:import"* />

<xs:field xpath=*"@namespace"* />

</xs:key>

</xs:element>

<xs:group name=*"anyTopLevelOptionalElement"*>

<xs:annotation>

<xs:documentation>

Any top level optional element allowed to appear more then once - any

child of definitions element except wsdl:types. Any extensibility

element is allowed in any place.

</xs:documentation>

</xs:annotation>

<xs:choice>

<xs:element name=*"import"* type=*"wsdl:tImport"* />

<xs:element name=*"types"* type=*"wsdl:tTypes"* />

<xs:element name=*"message"* type=*"wsdl:tMessage"*>

<xs:unique name=*"part"*>

<xs:selector xpath=*"wsdl:part"* />

<xs:field xpath=*"@name"* />

</xs:unique>

</xs:element>

<xs:element name=*"portType"* type=*"wsdl:tPortType"* />

<xs:element name=*"binding"* type=*"wsdl:tBinding"* />

<xs:element name=*"service"* type=*"wsdl:tService"*>

<xs:unique name=*"port"*>

<xs:selector xpath=*"wsdl:port"* />

<xs:field xpath=*"@name"* />

</xs:unique>

</xs:element>

</xs:choice>

</xs:group>

<xs:complexType name=*"tDefinitions"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleDocumented"*>

<xs:sequence>

<xs:group ref=*"wsdl:anyTopLevelOptionalElement"* minOccurs=*"0"*

maxOccurs=*"unbounded"* />

</xs:sequence>

<xs:attribute name=*"targetNamespace"* type=*"xs:anyURI"*

use=*"optional"* />

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"optional"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tImport"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleAttributesDocumented"*>

<xs:attribute name=*"namespace"* type=*"xs:anyURI"* use=*"required"* />

<xs:attribute name=*"location"* type=*"xs:anyURI"* use=*"required"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tTypes"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleDocumented"* />

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tMessage"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleDocumented"*>

<xs:sequence>

<xs:element name=*"part"* type=*"wsdl:tPart"* minOccurs=*"0"*

maxOccurs=*"unbounded"* />

</xs:sequence>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"required"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tPart"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleAttributesDocumented"*>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"required"* />

<xs:attribute name=*"element"* type=*"xs:QName"* use=*"optional"* />

<xs:attribute name=*"type"* type=*"xs:QName"* use=*"optional"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tPortType"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleAttributesDocumented"*>

<xs:sequence>

<xs:element name=*"operation"* type=*"wsdl:tOperation"*

minOccurs=*"0"* maxOccurs=*"unbounded"* />

</xs:sequence>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"required"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tOperation"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleDocumented"*>

<xs:sequence>

<xs:choice>

<xs:group ref=*"wsdl:request-response-or-one-way-operation"* />

<xs:group ref=*"wsdl:solicit-response-or-notification-operation"* />

</xs:choice>

</xs:sequence>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"required"* />

<xs:attribute name=*"parameterOrder"* type=*"xs:NMTOKENS"*

use=*"optional"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:group name=*"request-response-or-one-way-operation"*>

<xs:sequence>

<xs:element name=*"input"* type=*"wsdl:tParam"* />

<xs:sequence minOccurs=*"0"*>

<xs:element name=*"output"* type=*"wsdl:tParam"* />

<xs:element name=*"fault"* type=*"wsdl:tFault"* minOccurs=*"0"*

maxOccurs=*"unbounded"* />

</xs:sequence>

</xs:sequence>

</xs:group>

<xs:group name=*"solicit-response-or-notification-operation"*>

<xs:sequence>

<xs:element name=*"output"* type=*"wsdl:tParam"* />

<xs:sequence minOccurs=*"0"*>

<xs:element name=*"input"* type=*"wsdl:tParam"* />

<xs:element name=*"fault"* type=*"wsdl:tFault"* minOccurs=*"0"*

maxOccurs=*"unbounded"* />

</xs:sequence>

</xs:sequence>

</xs:group>

<xs:complexType name=*"tParam"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleAttributesDocumented"*>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"optional"* />

<xs:attribute name=*"message"* type=*"xs:QName"* use=*"required"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tFault"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleAttributesDocumented"*>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"required"* />

<xs:attribute name=*"message"* type=*"xs:QName"* use=*"required"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tBinding"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleDocumented"*>

<xs:sequence>

<xs:element name=*"operation"* type=*"wsdl:tBindingOperation"*

minOccurs=*"0"* maxOccurs=*"unbounded"* />

</xs:sequence>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"required"* />

<xs:attribute name=*"type"* type=*"xs:QName"* use=*"required"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tBindingOperationMessage"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleDocumented"*>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"optional"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tBindingOperationFault"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleDocumented"*>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"required"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tBindingOperation"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleDocumented"*>

<xs:sequence>

<xs:element name=*"input"* type=*"wsdl:tBindingOperationMessage"*

minOccurs=*"0"* />

<xs:element name=*"output"* type=*"wsdl:tBindingOperationMessage"*

minOccurs=*"0"* />

<xs:element name=*"fault"* type=*"wsdl:tBindingOperationFault"*

minOccurs=*"0"* maxOccurs=*"unbounded"* />

</xs:sequence>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"required"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tService"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleDocumented"*>

<xs:sequence>

<xs:element name=*"port"* type=*"wsdl:tPort"* minOccurs=*"0"*

maxOccurs=*"unbounded"* />

</xs:sequence>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"required"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name=*"tPort"*>

<xs:complexContent>

<xs:extension base=*"wsdl:tExtensibleDocumented"*>

<xs:attribute name=*"name"* type=*"xs:NCName"* use=*"required"* />

<xs:attribute name=*"binding"* type=*"xs:QName"* use=*"required"* />

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:attribute name=*"arrayType"* type=*"xs:string"* />

<xs:attribute name=*"required"* type=*"xs:boolean"* />

<xs:complexType name=*"tExtensibilityElement"* abstract=*"true"*>

<xs:attribute ref=*"wsdl:required"* use=*"optional"* />

</xs:complexType>

</xs:schema>

**Important parts of WSDL is given below.**

**<xs:element name=*"definitions"* type=*"wsdl:tDefinitions"*>**

**<xs:key name=*"message"*>**

**<xs:selector xpath=*"wsdl:message"* />**

**<xs:field xpath=*"@name"* />**

**</xs:key>**

**<xs:key name=*"portType"*>**

**<xs:selector xpath=*"wsdl:portType"* />**

**<xs:field xpath=*"@name"* />**

**</xs:key>**

**<xs:key name=*"binding"*>**

**<xs:selector xpath=*"wsdl:binding"* />**

**<xs:field xpath=*"@name"* />**

**</xs:key>**

**<xs:key name=*"service"*>**

**<xs:selector xpath=*"wsdl:service"* />**

**<xs:field xpath=*"@name"* />**

**</xs:key>**

**<xs:key name=*"import"*>**

**<xs:selector xpath=*"wsdl:import"* />**

**<xs:field xpath=*"@namespace"* />**

**</xs:key>**

**</xs:element>**

**<xs:sequence>**

**<xs:group ref=*"wsdl:anyTopLevelOptionalElement"* minOccurs=*"0"***

**maxOccurs=*"unbounded"* />**

**</xs:sequence>**

**It means types, message, portType, binding and service can occur many times in WSDL.**

**Schema for SOAP**

<xs:schema xmlns:xs=*"http://www.w3.org/2001/XMLSchema"*

xmlns:tns=*"http://schemas.xmlsoap.org/soap/envelope/"* targetNamespace=*"http://schemas.xmlsoap.org/soap/envelope/"*>

<!-- Envelope, header and body -->

<xs:element name=*"Envelope"* type=*"tns:Envelope"* />

<xs:complexType name=*"Envelope"*>

<xs:sequence>

<xs:element ref=*"tns:Header"* minOccurs=*"0"* />

<xs:element ref=*"tns:Body"* minOccurs=*"1"* />

<xs:any namespace=*"##other"* minOccurs=*"0"* maxOccurs=*"unbounded"*

processContents=*"lax"* />

</xs:sequence>

<xs:anyAttribute namespace=*"##other"* processContents=*"lax"* />

</xs:complexType>

<xs:element name=*"Header"* type=*"tns:Header"* />

<xs:complexType name=*"Header"*>

<xs:sequence>

<xs:any namespace=*"##other"* minOccurs=*"0"* maxOccurs=*"unbounded"*

processContents=*"lax"* />

</xs:sequence>

<xs:anyAttribute namespace=*"##other"* processContents=*"lax"* />

</xs:complexType>

<xs:element name=*"Body"* type=*"tns:Body"* />

<xs:complexType name=*"Body"*>

<xs:sequence>

<xs:any namespace=*"##any"* minOccurs=*"0"* maxOccurs=*"unbounded"*

processContents=*"lax"* />

</xs:sequence>

<xs:anyAttribute namespace=*"##any"* processContents=*"lax"*>

<xs:annotation>

<xs:documentation>

Prose in the spec does not specify that attributes are allowed on the

Body element

</xs:documentation>

</xs:annotation>

</xs:anyAttribute>

</xs:complexType>

<!-- Global Attributes. The following attributes are intended to be usable

via qualified attribute names on any complex type referencing them. -->

<xs:attribute name=*"mustUnderstand"*>

<xs:simpleType>

<xs:restriction base=*"xs:boolean"*>

<xs:pattern value=*"0|1"* />

</xs:restriction>

</xs:simpleType>

</xs:attribute>

<xs:attribute name=*"actor"* type=*"xs:anyURI"* />

<xs:simpleType name=*"encodingStyle"*>

<xs:annotation>

<xs:documentation>

'encodingStyle' indicates any canonicalization conventions followed in the contents

of the containing element. For example, the value

'http://schemas.xmlsoap.org/soap/encoding/' indicates the pattern

described in SOAP specification

</xs:documentation>

</xs:annotation>

<xs:list itemType=*"xs:anyURI"* />

</xs:simpleType>

<xs:attribute name=*"encodingStyle"* type=*"tns:encodingStyle"* />

<xs:attributeGroup name=*"encodingStyle"*>

<xs:attribute ref=*"tns:encodingStyle"* />

</xs:attributeGroup>

<xs:element name=*"Fault"* type=*"tns:Fault"* />

<xs:complexType name=*"Fault"* final=*"extension"*>

<xs:annotation>

<xs:documentation>Fault reporting structure</xs:documentation>

</xs:annotation>

<xs:sequence>

<xs:element name=*"faultcode"* type=*"xs:QName"* />

<xs:element name=*"faultstring"* type=*"xs:string"* />

<xs:element name=*"faultactor"* type=*"xs:anyURI"* minOccurs=*"0"* />

<xs:element name=*"detail"* type=*"tns:detail"* minOccurs=*"0"* />

</xs:sequence>

</xs:complexType>

<xs:complexType name=*"detail"*>

<xs:sequence>

<xs:any namespace=*"##any"* minOccurs=*"0"* maxOccurs=*"unbounded"*

processContents=*"lax"* />

</xs:sequence>

<xs:anyAttribute namespace=*"##any"* processContents=*"lax"* />

</xs:complexType>

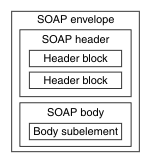
</xs:schema>

A SOAP message is an ordinary XML document containing the following elements −

* **Envelope −** Defines the start and the end of the message. It is a mandatory element.
* **Header −** Contains any optional attributes of the message used in processing the message, either at an intermediary point or at the ultimate end-point. It is an optional element.
* **Body −** Contains the XML data comprising the message being sent. It is a mandatory element.
* **Fault −** An optional Fault element that provides information about errors that occur while processing the message.

[**https://www-01.ibm.com/support/knowledgecenter/SSMKHH\_9.0.0/com.ibm.etools.mft.doc/ac55780\_.htm**](https://www-01.ibm.com/support/knowledgecenter/SSMKHH_9.0.0/com.ibm.etools.mft.doc/ac55780_.htm)

The following diagram shows the structure of a SOAP message.



<?xml version="1.0"?>

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope" SOAP-ENV:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<SOAP-ENV:Header>

...

</SOAP-ENV:Header>

<SOAP-ENV:Body>

...

<SOAP-ENV:Fault>

...

</SOAP-ENV:Fault>

...

</SOAP-ENV:Body>

</SOAP\_ENV:Envelope>

**The SOAP header**

The SOAP header (the <Header> element) is an optional sub-element of the SOAP envelope, and is used to pass application-related information that is processed by SOAP nodes along the message flow.

The immediate child elements of the header are called *header blocks*. A header block is an application-defined XML element, and represents a logical grouping of data which can be targeted at SOAP nodes that might be encountered in the path of a message from a sender to an ultimate receiver.

SOAP header blocks can be processed by SOAP intermediary nodes, and by the ultimate SOAP receiver node. However, in a real application, not every node processes every header block. Each node is typically designed to process particular header blocks, and each header block is processed by particular nodes.

The SOAP header enables you to add features to a SOAP message in a decentralized manner without prior agreement between the communicating parties. SOAP defines some attributes that can be used to indicate what can deal with a feature and whether it is optional or mandatory. Such control information includes, for example, passing directives or contextual information related to the processing of the message. This control information enables a SOAP message to be extended in an application-specific manner.

Although the header blocks are application-defined, SOAP-defined attributes on the header blocks indicate how the header blocks must be processed by the SOAP nodes. SOAP-defined attributes include:

**encodingStyle**

Indicates the rules used to encode the parts of a SOAP message. SOAP defines a narrower set of rules for encoding data than the flexible encoding that XML enables.

**actor (SOAP 1.1) or role (SOAP 1.2)**

In SOAP 1.2, the role attribute specifies whether a particular node will operate on a message. If the role specified for the node matches the role attribute of the header block, the node processes the header. If the roles do not match, the node does not process the header block. In SOAP 1.1, the actor attribute performs the same function.

Roles can be defined by the application, and are designated by a URI. For example, http://example.com/Log might designate the role of a node which performs logging. Header blocks that are processed by this node specify env:role="http://example.com/Log" (where the namespace prefix env is associated with the SOAP namespace name of http://www.w3.org/2003/05/soap-envelope).

The SOAP 1.2 specification defines three standard roles in addition to those which are defined by the application:

**http://www.w3.org/2003/05/soap-envelope/none**

None of the SOAP nodes on the message path should process the header block directly. Header blocks with this role can be used to carry data that is required for processing of other SOAP header blocks.

**http://www.w3.org/2003/05/soap-envelope/next**

All SOAP nodes on the message path are expected to examine the header block (provided that the header has not been removed by a node earlier in the message path).

**http://www.w3.org/2003/05/soap-envelope/ultimateReceiver**

Only the ultimate receiver node is expected to examine the header block.

**mustUnderstand**

This attribute is used to ensure that SOAP nodes do not ignore header blocks which are important to the overall purpose of the application. If a SOAP node determines, by using the **role** or **actor** attribute, that it should process a header block, the action taken depends on the value of the **mustUnderstand**attribute.

* 1 (SOAP 1.1) or true (SOAP 1.2): The node must either process the header block in a manner consistent with its specification, or not at all (and throw a fault).
* 0 (SOAP 1.1) or false (SOAP 1.2): The node is not obliged to process the header block.

In effect, the **mustUnderstand** attribute indicates whether processing of the header block is mandatory or optional.

**relay (SOAP 1.2 only)**

When a SOAP intermediary node processes a header block, the SOAP intermediary node removes the header block from the SOAP message. By default, the SOAP intermediary node also removes all header blocks that it ignored (because the **mustUnderstand** attribute had a value of false). However, when the relay attribute is specified with a value of true, the SOAP intermediary node retains the unprocessed header block in the message.

# The SOAP body

The SOAP body (the <Body> element) is a mandatory sub-element of the SOAP envelope, which contains information intended for the ultimate recipient of the message.

The body element and its associated child elements are used to exchange information between the initial SOAP sender and the ultimate SOAP receiver. SOAP defines one child element for the body: the <Fault> element, which is used for reporting errors. Other elements in the body are defined by the Web service that uses them.

# The SOAP fault

The SOAP fault (the <Fault> element) is a sub-element of the SOAP body, which is used for reporting errors.

If present, the SOAP fault element must appear as a body entry and must not appear more than once in a body element. The sub-elements of the SOAP fault element are different in SOAP 1.1 and SOAP 1.2.

## SOAP 1.1

In SOAP 1.1, the SOAP fault contains the following sub-elements:

**<faultcode>**

The<faultcode> element is a mandatory element in the <Fault> element. It provides information about the fault in a form that can be processed by software. SOAP defines a small set of SOAP fault codes covering basic SOAP faults, and this set can be extended by applications.

**<faultstring>**

The <faultstring> element is a mandatory element in the <Fault> element. It provides information about the fault in a form intended for a human reader.

**<faultactor>**

The <faultactor> element contains the URI of the SOAP node that generated the fault. A SOAP node that is not the ultimate SOAP receiver must include the <faultactor> element when it creates a fault; an ultimate SOAP receiver is not obliged to include this element, but might do so.

**<detail>**

The <detail> element carries application-specific error information related to the <Body> element. It must be present if the contents of the <Body> element were not successfully processed. The <detail> element must not be used to carry information about error information belonging to header entries. Detailed error information belonging to header entries must be carried in header entries.

## SOAP 1.2

In SOAP 1.2, the SOAP fault contains the following sub-elements:

**<Code>**

The <Code> element is a mandatory element in the <Fault> element. It provides information about the fault in a form that can be processed by software. It contains a <Value> element and an optional <Subcode> element.

**<Reason>**

The <Reason> element is a mandatory element in the <Fault> element. It provides information about the fault in a form intended for a human reader. The<Reason> element contains one or more <Text> elements, each of which contains information about the fault in a different language.

**<Node>**

The <Node> element contains the URI of the SOAP node that generated the fault. A SOAP node that is not the ultimate SOAP receiver must include the<Node> element when it creates a fault; an ultimate SOAP receiver is not obliged to include this element, but might do so.

**<Role>**

The <Role> element contains a URI that identifies the role in which the node was operating at the point the fault occurred.

**<Detail>**

The <Detail> element is an optional element, which contains application-specific error information related to the SOAP fault codes describing the fault. The presence of the <Detail> element has no significance as to which parts of the faulty SOAP message were processed.

# SOAP 1.1 and 1.2

SOAP is a lightweight, XML-based, protocol for exchange of information in a decentralized, distributed environment.

The protocol consists of three parts:

* An envelope that defines a framework for describing what is in a message and how to process it.
* A set of encoding rules for expressing instances of application-defined data types.
* A convention for representing remote procedure calls and responses.

SOAP can be used with other protocols, such as HTTP.

The specifications for SOAP are published by the World Wide Web Consortium (W3C).

* [World Wide Web Consortium (W3C)](http://www.w3.org/)

The specification for SOAP 1.1 is described in:

* [Simple Object Access Protocol 1.1](http://www.w3.org/TR/soap/)

This specification has not been endorsed by the W3C, but forms the basis for the SOAP 1.2 specification. The specification for SOAP 1.1 expands the SOAP acronym to Simple Object Access Protocol.

SOAP 1.2 is a W3C recommendation and is published in two parts:

[Part 1: Messaging Framework](http://www.w3.org/TR/soap12-part1/).

[Part 2: Adjuncts](http://www.w3.org/TR/soap12-part2/).

The specification also includes a primer that is intended to provide a tutorial on the features of the SOAP Version 1.2 specification, including usage scenarios. The specification for SOAP 1.2 does not expand the acronym. The primer is published at:

**Soap Security Header**

[**https://docs.oracle.com/cd/E21455\_01/common/tutorials/authn\_ws\_user.html**](https://docs.oracle.com/cd/E21455_01/common/tutorials/authn_ws_user.html)

|  |
| --- |
| **WS-Security Username Authentication** |

|  |
| --- |
| **Contents** |
| |  | | --- | | * [Overview](https://docs.oracle.com/cd/E21455_01/common/tutorials/authn_ws_user.html#p_authn_ws_user_over) * [General Configuration](https://docs.oracle.com/cd/E21455_01/common/tutorials/authn_ws_user.html#p_authn_ws_user_conf_general) * [Token Validation](https://docs.oracle.com/cd/E21455_01/common/tutorials/authn_ws_user.html#p_authn_ws_user_conf_token) * [Token Verification via Repository](https://docs.oracle.com/cd/E21455_01/common/tutorials/authn_ws_user.html#p_authn_ws_user_conf_repository) | |

|  |
| --- |
|  |
| |  | | --- | | **Overview** | | A WS-Security *Username Token* enables an end-user identity to be passed over multiple hops before reaching the destination Web Service. The user identity is inserted into the message and is available for processing at each hop on its path.  The client user name and password are encapsulated in a WS-Security <wsse:UsernameToken>. When the Enterprise Gateway receives this token, it can perform one of the following tasks, depending on the requirements:   * Ensure that the timestamp on the token is still valid * Authenticate the user name against a repository * Authenticate the user name and password against a repository   The following sample SOAP message contains two <wsse:UsernameToken> blocks:   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | |  | <?xml version="1.0" encoding="iso-8859-1"?>  <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">  <soap:Header>  <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext">  <wsse:UsernameToken wsu:Id="sample"  xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">  <wsse:Username>sample</wsse:Username>  <wsse:Password Type="wsse:PasswordText">oracle</wsse:Password>  <wsu:Created>2004-05-19T08:44:51Z</wsu:Created>  </wsse:UsernameToken>  </wsse:Security>  <wsse:Security soap:actor="oracle"  xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext">  <wsse:UsernameToken wsu:Id="oracle"  xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">  <wsse:Username>oracle</wsse:Username>  <wsse:Password Type="wsse:PasswordText">oracle</wsse:Password>  <wsu:Created>2004-05-19T08:46:04Z</wsu:Created>  </wsse:UsernameToken>  </wsse:Security>  </soap:Header>  <soap:Body>  <getHello xmlns="http://www.oracle.com"/>  </soap:Body>  </soap:Envelope> |  | |  |  |  | |   This topic explains how to configure the Enterprise Gateway to authenticate users using a WS-Security <wsse:UsernameToken>. |      |  | | --- | | **General Configuration** | | To configure general settings, complete the following fields:  **Name:**  Enter an appropriate name for this filter.  **Actor:**  The example SOAP message at the top of this page contains two <wsse:UsernameToken> blocks. You must specify which block contains the <wsse:UsernameToken> used to authenticate the end-user. Specify the SOAP Actor/Role of the WS-Security block that contains the token.  **Credential Format:**  The Enterprise Gateway can authenticate users against a user profile repository based on User Names, X.509 Distinguished Names, or email addresses. Unfortunately, the WS-Security specification does not provide a means of specifying the type of <wsse:UsernameToken>, and so it is necessary for the administrator to do so using the **Credential Format** field. The type specified here is used internally by the Enterprise Gateway in subsequent authorization filters. |      |  | | --- | | **Token Validation** | | Each wsse:UsernameToken contains a timestamp inserted into the <wsu:Created> element. Using this timestamp together with the details entered in this section, the Enterprise Gateway can determine whether the WS-SecurityUsernameToken has expired. The <wsu:Created> element is as follows:   |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | |  |  |  | |  | <wsse:UsernameToken wsu:Id="oracle"  xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">  <wsu:Created>2006.01.13T-10:42:43Z</wsu:Created>  ...  </wsse:UsernameToken> |  | |  |  |  | |   To configure token validation settings, complete the following fields:  **Drift Time:**  Specified in seconds to account for differences in the clock times between the machine on which the token was generated and the machine running the Enterprise Gateway. Using the *start time*, *end time*, and *drift time*, the token is considered valid if the current time falls between the following times:   |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | |  |  |  | |  | [start - drift] and [start + drift + end] |  | |  |  |  | |   **Validity Period:**  Specifies the lifetime of the token, where the value of the <wsu:Created> element represents the *start time* of the assertion, and the time period entered represents the *end time*.  **Timestamp Required:**  Select this option if you want to ensure that the Username Token contains a timestamp. If no timestamp is found in the Username Token, a SOAP Fault is returned.  **Nonce Required:**  Select this option to ensure that the Username Token contains a <wsse:Nonce> element. You can use the combination of a timestamp and a nonce to help prevent replay attacks. |      |  | | --- | | **Token Verification via Repository** | | Having validated the timestamp on the token, the Enterprise Gateway can then optionally authenticate the user name and password contained in the token. The following options are available:   * **No Verification**  No verification of the user name and password is performed. Only the timestamp on the token is validated. This is the default behavior. * **Verify Username Only**  Only the user name is looked up in the selected repository. If the user name is found in this repository, the user is authenticated. Select the **No password allowed** checkbox to block messages that contain a Username Token with a <wsse:Password> element. * **Verify Username and Password**  The user name is looked up in the selected repository and is only authenticated if the corresponding password matches the one configured in the repository. If you select this option, you must select the type of the password. Both cleartext and digest formats are supported. Select the appropriate option.   **Repository Name:**  The Enterprise Gateway attempts to authenticate users against the selected **Authentication Repository**. User profiles can be stored in the local store, a database, or an LDAP directory. For details on adding a new repository, and editing or deleting a repository, see the [Authentication Repository](https://docs.oracle.com/cd/E21455_01/common/tutorials/common_user_store.html) tutorial.  **Remove enclosing WS-Security element on successful validation:**  Select this option if you wish to remove the WS-Security block that contains the Username Token after the token has been successfully authenticated. For example, in the above sample SOAP message that contains two<wsse:UsernameToken> elements in two different WS-Security blocks, you could configure the Enterprise Gateway to remove one of these on successful authentication. | |

# Understanding WS-Security

[**https://msdn.microsoft.com/en-us/library/ms977327.aspx**](https://msdn.microsoft.com/en-us/library/ms977327.aspx)

**Summary:**This article looks at how to use WS-Security to embed security within the SOAP message itself, exploring the concerns WS-Security addresses: authentication, signatures, and encryption. (14 printed pages)

#### **Contents**

[Introduction](https://msdn.microsoft.com/en-us/library/ms977327.aspx#understw_topic1) [Parallels in Daily Life](https://msdn.microsoft.com/en-us/library/ms977327.aspx#understw_topic2) [Applying Existing Concepts to SOAP Messages](https://msdn.microsoft.com/en-us/library/ms977327.aspx#understw_topic3) [WS-Security SOAP Header](https://msdn.microsoft.com/en-us/library/ms977327.aspx#understw_topic4) [Conclusion](https://msdn.microsoft.com/en-us/library/ms977327.aspx#understw_topic5)  
[Resources](https://msdn.microsoft.com/en-us/library/ms977327.aspx#understw_topic6)

## Introduction

Before I explain what WS-Security is, I believe that it is important to understand why WS-Security exists at all. Many people new to Web services see SOAP as a way to exchange messages between two endpoints over HTTP. Over HTTP, one can authenticate the caller, sign the message, and encrypt the contents of the message. This makes the message secure in several dimensions: the caller is known, the receiver of the message can verify that the message did not change in transit, and entities watching the wire traffic cannot figure out what data is being exchanged. For those looking at SOAP messaging to solve bigger problems, however, HTTP-based security simply isn't enough. Many of the bigger problems involve sending the message along a path more complicated than request/response or over a transport that does not involve HTTP. The identity, integrity, and security of the message and the caller need to be preserved over multiple hops. More than one encryption key may be used along the route. Trust domains will be crossed. HTTP and its security mechanisms only address point-to-point security. More complex solutions need end-to-end security baked in. WS-Security addresses how to maintain a secure context over a multi-point message path.

**Note**   This article assumes that you are already familiar with [XML Canonicalization](http://www.w3.org/tr/xml-exc-c14n/), [XML Signature](http://www.w3.org/signature/), and[XML Encryption](http://www.w3.org/encryption/2001/).

WS-Security addresses security by leveraging existing standards and specifications. This avoids the necessity to define a complete security solution within WS-Security. The industry has solved many of these problems. Kerberos and X.509 address authentication. X.509 also uses existing PKI for key management. XML Encryption and XML Signature describe ways of encrypting and signing the contents of XML messages. XML Canonicalization describes ways of making the XML ready to be signed and encrypted. What WS-Security adds to existing specifications is a framework to embed these mechanisms into a SOAP message. This is done in a transport-neutral fashion.

WS-Security defines a SOAP Header element to carry security-related data. If XML Signature is used, this header can contain the information defined by XML Signature that conveys how the message was signed, the key that was used, and the resulting signature value. Likewise, if an element within the message is encrypted, the encryption information such as that conveyed by XML Encryption can be contained within the WS-Security header. WS-Security does not specify the format of the signature or encryption. Instead, it specifies how one would embed the security information laid out by other specifications within a SOAP message. WS-Security is primarily a specification for an XML-based security metadata container.

What does WS-Security do, beyond leveraging other existing protocols for message authentication, integrity, and privacy? It specifies a mechanism for transferring simple user credentials via the **UsernameToken** element. To send binary tokens that were used for encryption or signing the message, a **BinarySecurityToken** is also defined. Within this header, messages can store information about the caller, how the message was signed, and how the message was encrypted. WS-Security presents an end-to-end solution for Web service security by keeping all security information in the SOAP part of the message.

In this article, we will take a look at how to use WS-Security and friends to embed security within the SOAP message itself. We will look at the concerns WS-Security addresses:

* Authentication
* Signatures
* Encryption

This triad addresses the main concerns of security and answers the questions:

* Who am I authorizing?
* Was the message modified between hops?
* Did this message come from whom I think it came from?
* How do I hide things I only want certain parties to see?

To begin with, let's look at some analogies seen in every day life.

## Parallels in Daily Life

To understand what WS-Security is trying to do, I first want to take a look at a real-world parallel. Specifically, when and how do you use credentials in everyday life? After all, in day-to-day living you use credentials all the time. If someone asks you to prove your age, you dig into your wallet and pull out a driver's license. When you go to pay for an item without using currency, a credit card is used to identify you to the credit agency. When crossing a country's border or while in a foreign country, a passport vouches for your identity. All of these items are credentials. They assert that the owner of the credit card, driver's license, or passport is the person named on the document. They do not authenticate your identity, though. Authentication is an action that a document cannot perform. In the world of paper documents, people perform authentication. How does that side of authentication work?

When you present a driver's license or passport, a person reading the documents performs a few different actions to verify that the documents are real and that you are the rightful owner of those documents:

* Both documents contain a picture of the registered holder of the document as well as other identifying characteristics: height, weight, and eye color. The person reading the document can make sure you look like the person shown and described by the document.
* The documents expire on a regular basis. This is done so that up to date descriptive data is on the document.
* The documents contain a signature that can be compared against the signature of the person presenting the documents. The difficulty of accurately reproducing another person's signature makes this a reasonable way to check identity when used in combination with the description of the person.

These documents have other important properties as well. They have marking that allow someone to quickly verify that the documents are genuine. The documents themselves are granted by organizations that we trust to provide valid identity information: local and national governments. I mentioned credit cards too. These are different from a driver's license or passport.

* They are issued by banks, not governments.
* They only contain a signature and a name to identify the card holder.
* They can only be used to verify identity in the presence of another supporting document such as a government issued ID.

What does something like a credit card do then?

Credit cards typically rely on only using a signature for authentication. Some cards include photos to make authentication a little more solid. Because of the weak authentication provided by credit cards, many establishments will ask to see a government issued ID with the credit card. In terms of security, when you present the credit card, you assert that you have the right to charge goods and services and that the organization that gave you the credit card will pay the merchant. The merchant can validate your identity as the valid cardholder by comparing your government-issued photo ID to your physical person. (Of course, if you perform the transaction over the phone or the Internet, this part of my argument falls apart, but it suffices to state that other mechanisms exist to protect you in these arenas as well.)

## Applying Existing Concepts to SOAP Messages

WS-Security seeks to move a lot of these concepts about identification and authorization into the world of SOAP messaging. In order to do something meaningful with a SOAP message, that message must contain information that does the following things:

* Identify the entity or entities involved with the message.
* Prove that the entities have the correct group memberships.
* Prove that the entities have the correct set of access rights.
* Prove that the message has not changed.

Finally, we also want a mechanism that would hide information from unauthorized parties. In the world of personal identification, I prove who I am with my driver's license or passport. I prove that I have certain rights through membership cards. In my wallet, I have cards that allow me to charge goods and services, check out books from the library, direct medical bills to my insurance provider, and receive discounts at local grocery stores. WS-Security allows me to apply the same concepts to SOAP messages. Using security tokens to identify the caller and assert its rights, a message could convey the following information:

* Caller identity: I am Joe User.
* Group membership: I am a ColdRooster.com developer.
* Rights assertions: Because I am a ColdRooster.com developer, I can create databases and add Web applications to the ColdRooster.com machines.

To create a message that can create a new database on the ColdRooster.com servers using an authentication technology such as Kerberos, the application would have to acquire a number of security tokens. To start with, the application creating the message would need to acquire a security token that identifies it as acting on behalf of Joe User. Joe User provides that token when he logs in via a username/password or by using a smart card. Assuming that the security infrastructure uses Kerberos, the environment Joe is using has a Key Distribution Center that grants Joe a Ticket Granting Ticket (TGT) when he logs in. When Joe decides to create a new database on ColdRooster.com, the environment goes to a Ticket Granting Service and requests a Service Ticket that shows that Joe has the right to create a new database on ColdRooster.com. The environment takes that Service Ticket (ST) and presents it to the database server at ColdRooster.com. That database server validates the ticket and then allows Joe to create the new process.

WS-Security seeks to encapsulate the security interactions described above within a set of SOAP Headers. WS-Security handles credential management in two ways. It defines a special element, **UsernameToken**, to pass the username and password if the Web service is using custom authentication. WS-Security also provides a place to provide binary authentication tokens such as Kerberos Tickets and X.509 Certifications: BinarySecurityToken.

Figure 1 depicts what will become a fairly common message flow.

**Figure 1. Typical message flow.**

The Security Token service might be Kerberos, PKI, or a username/password validation service. This service may not be Web service-based. Indeed, a Kerberos service ticket granting service might be accessed through the Kerberos protocols using operating system security functions. Once the client gets the tokens it wants to use in the message, the client will embed those tokens within the message. The client should sign the message with a piece of data that only they know. The server will be able to deduce the signature in a number of ways. If the client is using a **UsernameToken** for authentication, the client should send a hashed password and sign the message using that password. The server will be able to verify that the client sent the message if the signatures it generates for the message match the signatures contained in the message.

When using X.509 certificates, the message can be signed using the private key. The message should contain the certificate in a **BinarySecurityToken**. When using X.509, anyone who knows the X.509 public key can verify the signature. Finally, when using Kerberos, the message could be signed or encrypted with a session key embedded in the Kerberos ticket. Because the Kerberos ticket will be keyed for the receiver of the token, only the receiver will be able to decrypt the ticket, discover the session key, and verify the signature.

It is critical that SOAP messages be signed or encrypted if authentication is important. Why? It isn't enough that a valid identity token is added to a message. These tokens can be lifted from a valid message and added to messages used by attackers. There needs to be evidence that the identity used in the message created the message. Without using XML Signature and signing the message, you cannot tell that the message has not been changed or that the identity token has not been abused.

At this point, I think you understand what WS-Security does. Let's dig a little deeper and look at how.

## WS-Security SOAP Header

Starting in this section and continuing throughout the rest of the article, I will be using XML snippets. So that I don't have to show XML Namespace declarations all over and muddy the snippets, I will use the following XML Namespaces:

**Table 1: XML Namespaces**

|  |  |  |
| --- | --- | --- |
| Namespace | Description | Namespace URI |
| Xs | XML Schema | http://www.w3.org/2001/XMLSchema |
| Wsse | WS-Security | http://schemas.xmlsoap.org/ws/2002/07/secext |
| Wsu | Utility elements | http://schemas.xmlsoap.org/ws/2002/07/utility |
| Soap | SOAP elements | http://schemas.xmlsoap.org/soap/envelope/ |

The WS-Security specification defines a new SOAP header. To understand what the WS-Security SOAP header contains, I think it would be helpful to look at the schema fragment for the element first.

<xs:element name="Security">

<xs:complexType>

<xs:sequence>

<xs:any processContents="lax"

minOccurs="0" maxOccurs="unbounded">

</xs:any>

</xs:sequence>

<xs:anyAttribute processContents="lax"/>

</xs:complexType>

</xs:element>

As you can see, the Security header element allows any XML element or attribute to live within it. This allows the header to adapt to whatever security mechanisms your application needs. If this sounds a little odd, think about how the SOAP header and body work. The header and body both can contain a collection of XML elements. The SOAP specification makes few claims about the contents of these elements other than the fact that they cannot contain XML processing instructions.

WS-Security needs this type of structure because of what the header must do. It must be able to carry multiple security tokens to identify the caller's rights and identity. If the message is signed, the header must contain information about how it was signed and where the information regarding the key is stored. The key may be in the message or stored elsewhere and merely referenced. Finally, information about encryption must also be able to be carried in this header.

So, how does an intermediary know which WS-Security header it owns? A SOAP message may contain multiple WS-Security headers. Each header is identified by a unique actor. No two WS-Security headers can use the same actor or omit the actor. This makes it easy for intermediaries to identify which WS-Security headers contain the information they need. Of course, the intermediary does need to know which actor URI it handles. Associating a URI with an actor and making sure that the intermediary knows what to do is something that must be handled via programming. The actor attribute in any SOAP header is meant to say "this header is meant for any endpoint acting in the capacity indicated by the actor URI." What gives that URI meaning? The team that architects the Web service gives meaning to the URI. This means that an intermediary may act in varying capacities. As a result, that intermediary may consume zero, one, or more headers. Yes, it may even consume multiple security headers.

### **WS-Security Addendum**

After evaluating WS-Security for a little while, a number of items came out that needed to be made clearer for security in particular. Also, additional items needed to be specified for Web services in general. The parts of the addendum that apply to security are covered throughout the article. In this section, I want to take a look at two items that are not specific to security: **wsu:Id**and **wsu:Timestamp**. The addendum specifies exactly what these two items do and how they should be used.

#### **wsu:Id**

The **Id** attribute uses the XML Schema ID type. This element was added to simplify processing for Web service intermediaries as well as receivers. The value of this attribute must not be duplicated elsewhere in the document. The addendum does not go into more detail about how the element should be used other than as a unique identifier in GXA specifications. The door is left wide open to allow other specifications to restrict the usage of **Id**.

#### **wsu:Timestamp**

A common concern in message-oriented systems relates to the timeliness of data. If the data is too old, it may get thrown out. If two contradicting messages arrive, the related timestamps may be used to decide which message gets executed and which one is ignored. To handle the time-related issues that showed up in WS-Security and that will show up in other GXA specifications, the **wsu:Timestamp**element, along with a few helper elements, was defined.

The interesting events in a message's life are the time it was created, the time the sender wants the message to expire, and the time that the message was received. By knowing the creation and expiration time, a receiver can decide if the data is new enough for its own use or if the data has become so stale that the message should be discarded. The elements that convey this data are:

* **wsu:Created**: Contains the time that the message was created.
* **wsu:Expires**: Set by a sender or intermediary, this identifies when the message expires.
* **wsu:Received**: Explains when the message was received by a particular intermediary.

All of the above elements may appear independently or as part of a **wsu:Timestamp**element. Each may contain a**wsu:Id** attribute to uniquely identify the item. By default, these timestamps express the time as an **xs:dateTime** type. To allow flexibility for other, nonstandard time stamps that may be meaningful in other problem domains, each of these items also contains an attribute named **ValueType**. This attribute does not need to appear if the time is expressed as**xs:dateTime**.

The **wsu:Received** element allows for two extra attributes not found on **wsu:Created** or **wsu:Expires**. The element can express the URI of the actor it is related to, using the **Actor** attribute and the amount of delay, in milliseconds, caused by the actor using the **Delay** attribute.

As I mentioned, you can use the **wsu:Received**, **wsu:Created**, and **wsu:Expires**elements within other structures. For example, it may be common to see the **wsu:Created**element to indicate when a particular element was added to the message. When indicating more information about a message and using more than one of these elements at a time, the elements can be wrapped inside of a **wsu:Timestamp** element. Each of three elements may only appear once within a timestamp. The timestamp may used on the message as a whole, in which case it appears as a child of the **soap:Header**node. For example, a message could indicate it was valid for the next five minutes using the following **wsu:Timestamp**header.

<wsu:Timestamp>

<wsu:Created wsu:Id=

"Id-2af5d5bd-1f0c-4365-b7ff-010629a1256c">

2002-08-19T16:15:31Z

</wsu:Created>

<wsu:Expires wsu:Id=

"Id-4c337c65-8ae7-439f-b4f0-d18d7823e240">

2002-08-19T16:20:31Z

</wsu:Expires>

</wsu:Timestamp>

At this point, I believe that you have enough background information to dig into how authentication, signing, and encryption work with WS-Security.

### **Authentication**

WS-Security provides for an infinite number of ways to validate a user. The specification addresses three methods from that infinite number:

* Username/Password
* PKI through X.509 Certificates
* Kerberos

In this section, we will look at how each of these authentication methods works and how that information is encoded into a SOAP message.

#### **Username/Password**

One of the most common ways to pass around caller credentials is to use a username and password combination. This is a technique used in HTTP Basic and Digest authentication. As a matter of fact, if you are familiar with how HTTP Digest authentication works, you will feel right at home with this authentication mechanism. To pass user credentials in this manner, WS-Security has defined the **UsernameToken** element. Schema for the element is as follows:

<xs:element name="UsernameToken">

<xs:complexType>

<xs:sequence>

<xs:element ref="Username"/>

<xs:element ref="Password" minOccurs="0"/>

</xs:sequence>

<xs:attribute name="Id" type="xs:ID"/>

<xs:anyAttribute namespace="##other"/>

</xs:complexType>

</xs:element>

This schema fragment references two other types: **Username** and **Password**. These two types are essentially strings that may contain extra attributes as needed. **Password** contains an attribute named **Type** that indicates how the password is being passed around. A password can be passed as plain text or in digest format. When passing a **UsernameToken** in a SOAP message, the XML may come across as something like this:

<wsse:UsernameToken>

<wsse:Username>scott</wsse:Username>

<wsse:Password Type="wsse:PasswordText">password</wsse:Password>

</wsse:UsernameToken>

What you see here is an example of the password being sent as plain text. This particular solution looks pretty easy to break. If you want the password sent in a more secure manner, you can send it is as a digest hash.

<wsse:UsernameToken>

<wsse:Username>scott</wsse:Username>

<wsse:Password Type="wsse:PasswordDigest">

KE6QugOpkPyT3Eo0SEgT30W4Keg=</wsse:Password>

<wsse:Nonce>5uW4ABku/m6/S5rnE+L7vg==</wsse:Nonce>

<wsu:Created xmlns:wsu=

"http://schemas.xmlsoap.org/ws/2002/07/utility">

2002-08-19T00:44:02Z

</wsu:Created>

</wsse:UsernameToken>

This adds a bit more security because the password is now obscured using a SHA1 hash. The password digest is the concatenation of the nonce plus the creation time plus the password. The nonce is 16 bytes long and is passed along as a base64 encoded value. The way this works is that the client creates the password hash using all of this information plus the password. The receiver verifies the data by getting the plain password and creating the hash again. If the results agree, the password must be correct. This protection does not protect against replay attacks. If you use it, make sure to also include a **wsu:Timestamp** header with a small enough time window for the created and expired values. Then, sign the **wsu:Timestamp** elements within the message so that any tampering with the timestamp can be detected. Otherwise, an attacker could use the complete **UsernameToken** to attack your Web service. To defend against a replay attack, you will also need to include a mechanism that tracks some unique characteristic of the incoming messages. This mechanism needs to save this characteristic in a cache for at least the timeout period of the message.

#### **X.509 Certificates**

Another option to use when authenticating users is to simply send around an X.509 certificate. An X.509 certificate tells you exactly who the user is. Using PKI, you can map the certificate to an existing user in your application. Using the certificate on its own would make for some pretty easy replay attacks. As a result, it's a good idea to force the message sender to also sign the message using their private key. That way, when the message gets decrypted, you'll know it is really the user.

When a message does send along an X.509 certificate, it will pass the public version of the certificate in a WS-Security token named **BinarySecurityToken**. The certificate itself gets sent along as base64 encoded data. The**BinarySecurityToken** has the following schema:

<xs:element name="BinarySecurityToken">

<xs:complexType>

<xs:simpleContent>

<xs:extension base="xs:string">

<xs:attribute name="Id" type="xs:ID" />

<xs:attribute name="ValueType" type="xs:QName" />

<xs:attribute name="EncodingType" type="xs:QName" />

<xs:anyAttribute namespace="##other"

processContents="strict" />

</xs:extension>

</xs:simpleContent>

</xs:complexType>

</xs:element>

At its most basic, this item contains a string, a unique identifier, and some information indicating what type of value is included and how it was encoded. The **ValueType** may be any of the following values, defined by the **ValueTypeEnum**in the WS-Security schema document:

* **wsse:X509v3**: An X.509, version 3 certificate.
* **wsse:Kerberosv5TGT**: A ticket granting ticket as defined by section 5.3.1 of the Kerberos specification.
* **wsse:Kerberosv5ST**: A service ticket as defined by section 5.3.1 of the Kerberos specification.

If this information on Kerberos doesn't make any sense to you, I'll explain it a little better in the next section. The**EncodingType** is another enumeration. If it is set to wsse:Base64Binary or wsse:HexBinary. As you might guess, this value simply indicates which encoding method was used. In a WS-Security header, this element would look something like this when passing an X.509 certificate:

<wsse:BinarySecurityToken

ValueType="wsse:X509v3"

EncodingType="wsse:Base64Binary"

Id="SecurityToken-f49bd662-59a0-401a-ab23-1aa12764184f"

>MIIHdjCCB...</wsse:BinarySecurityToken>

Remember, when you use an X.509 certificate, you want to do something else as well, such as sign the message. The signature, created using the certificate's private key, proves that the client is the rightful owner of the certificate. Such a message could be replayed. To help mitigate the problems around replays, you would institute a policy that states how old a message can be before it is ignored. The time should travel in a **wsu:Timestamp** element that is shipped as a SOAP Header within the message.

#### **Kerberos**

To use Kerberos, a user presents a set of credentials such as username/password or an X.509 certificate. If everything checks out, the security system grants the user a ticket granting ticket (TGT). The TGT is an opaque piece of data that the user cannot read but must present in order to access other resources. The user will typically present the TGT in order to get a service ticket (ST). The way the system works is as follows:

1. A client authenticates to a Key Distribution Center (KDC) and is granted a TGT.
2. The client takes the TGT and uses it to access a Ticket Granting Service (TGS).
3. The client requests an ST for a particular network resource. The TGS then issues the ST to the client.
4. The client presents the ST to the network resource and begins accessing the resource with the permissions the ST indicated.

Kerberos is appealing because it contains a mechanism for the client to prove their identity to a service and for the service to prove their identity to the client. The ST is only good for accessing the one network resource and can be used to discover who the caller is. When presenting a Kerberos ticket in a message, the data needs to be blindly copied into the message itself. WS-Security does not explain how a TGT or ST is obtained.

### **Signing**

When a message is signed, it is nearly impossible to tamper with the message. Message signing does not protect the message itself from external parties seeing its contents. Using the signature, the receiver of the SOAP message can know that the signed elements have not changed en route. You should use XML Signature to sign messages whenever possible. Why? XML Signature already handles a number of the tougher items to figure out. WS-Security simply explains how to use signing to prove that the message has not changed. All three of the authentication mechanisms mentioned above provides a way to sign the message so that you can be sure of two things:

* The user identified by the X.509 certificate, **UsernameToken**, or Kerberos ticket signed the message.
* The message has not been tampered with since it was signed.

Each of the methods provides a secret that can be used to sign the message. X.509 allows the sender to sign the message using their private key. Kerberos provides a session key that the sender creates and transmits in the ticket. Only the intended receiver of that message can read the ticket, discover the session key, and verify the authenticity of the signature. Finally, the **UsernameToken** could be signed using the password.

The signature is generated using XML Signature. To sign a simple message such as "Hello World," almost every element in the message needs to be individually signed. **wsu:Timestamp**presents an interesting problem because an intermediary may add a **wsu:Received**element to **wsu:Timestamp**. Every time an element changes, the signature needs to be updated or else things won't look right. Why? If the content changes, the signature should not match. Within a SOAP Message, the signatures and required extra data add quite a bit of extra information.

### **Encryption**

There are times when proving the message sender's identity and showing that the message was not changed is not enough. If you send a credit card number or bank account number in a plain-text but signed manner, an attacker can actually verify that no other attackers have changed the contents of the message. As a result, they have a high confidence that the data is valid. That's no good for you, is it? Instead, you'd like the data encrypted in such a way that only the intended message recipient can read the message. Anyone watching the wire exchange should remain oblivious to the contents of the message. As with signing messages, the WS-Security specification does the right thing and adopts a standard that already exists and does the job of encryption well. That's right, they incorporated XML Encryption.

When you encrypt data, you can choose to use either symmetric or asymmetric encryption. Symmetric encryption requires a shared secret. That is, the key that is used to encrypt the message is the same key that you would use to decrypt the message. Symmetric encryption is good if you control both endpoints and can trust the people and applications that use the keys. Symmetric encryption does have a problem with key distribution. At some point in time, the key needs to be sent to the receiver. How do you do this? Do you ship a disk in the mail or negotiate the key when it is needed? Both options will work.

If you need to send data using easily distributed keys, look to asymmetric encryption. X.509 certificates allow for this. The endpoint receiving the data can publicly post its certificate and allow anyone and everyone to encrypt information using the public key. Only the receiver will know the private key. Because of this, only the receiver can take the encrypted data and turn it back into something readable.

So, what would an encrypted message look like? If you are using Triple-DES, both the sender and receiver would have to exchange the key in some secure manner. The symmetric key can be hidden inside a Kerberos ticket, or exchanged out of band. The WS-Security-based message with embedded XML Encryption information would look something like this:

<?xml version="1.0" encoding="utf-8" ?>

<soap:Envelope

xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"

xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">

<soap:Header

xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext"

xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">

<wsu:Timestamp>

<wsu:Created

wsu:Id="Id-3beeb885-16a4-4b65-b14c-0cfe6ad26800"

>2002-08-22T00:26:15Z</wsu:Created>

<wsu:Expires

wsu:Id="Id-10c46143-cb53-4a8e-9e83-ef374e40aa54"

>2002-08-22T00:31:15Z</wsu:Expires>

</wsu:Timestamp>

<wsse:Security soap:mustUnderstand="1" >

<xenc:ReferenceList>

<xenc:DataReference

URI="#EncryptedContent-f6f50b24-3458-41d3-aac4-390f476f2e51" />

</xenc:ReferenceList>

<xenc:ReferenceList>

<xenc:DataReference

URI="#EncryptedContent-666b184a-a388-46cc-a9e3-06583b9d43b6" />

</xenc:ReferenceList>

</wsse:Security>

</soap:Header>

<soap:Body>

<xenc:EncryptedData

Id="EncryptedContent-f6f50b24-3458-41d3-aac4-390f476f2e51"

Type="http://www.w3.org/2001/04/xmlenc#Content">

<xenc:EncryptionMethod Algorithm=

"http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />

<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

<KeyName>Symmetric Key</KeyName>

</KeyInfo>

<xenc:CipherData>

<xenc:CipherValue

>InmSSXQcBV5UiT... Y7RVZQqnPpZYMg==</xenc:CipherValue>

</xenc:CipherData>

</xenc:EncryptedData>

</soap:Body>

</soap:Envelope>

The preceding message contains information about what data was encrypted as well as how the encryption was performed. For anyone who does not have access to the key, the cipher text inside the **soap:Body** cannot be decrypted.

When performing asymmetric encryption, the private key needs to be known to the receiver of the message in order to decrypt that message. Exchanging the public key has to be figured out ahead of time.

## Conclusion

WS-Security allows for a SOAP message to identify the caller, sign the message, and encrypt message contents. Whenever possible, existing specifications are reused to reduce the amount of invention required to securely deliver a SOAP message. Because all of the information is delivered within the message itself, the message becomes transport neutral. The message would be secure if it was delivered by HTTP, e-mail, or on CD-ROM.

## Resources

* [Canonicalization](http://www.w3.org/tr/xml-exc-c14n/)
* [Signature](http://www.w3.org/signature/)
* [Encryption](http://www.w3.org/encryption/2001/)
* [WS-Security Specification](https://msdn.microsoft.com/en-us/library/ms951257.aspx)
* [WS-Security Addendum](https://msdn.microsoft.com/en-us/library/ms951252.aspx)

# XML Schema any Element

[**http://www.w3schools.com/schema/el\_any.asp**](http://www.w3schools.com/schema/el_any.asp)

## Definition and Usage

The any element enables the author to extend the XML document with elements not specified by the schema.

### **Element Information**

* **Parent elements:** choice, sequence

### **Syntax**

<any  
id=ID  
maxOccurs=nonNegativeInteger|unbounded  
minOccurs=nonNegativeInteger  
namespace=namespace  
processContents=lax|skip|strict  
*any attributes*  
>  
  
(annotation?)  
  
</any>

(The ? sign declares that the element can occur zero or one time inside the any element)

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| id | Optional. Specifies a unique ID for the element |
| maxOccurs | Optional. Specifies the maximum number of times the any element can occur in the parent element. The value can be any number >= 0, or if you want to set no limit on the maximum number, use the value "unbounded". Default value is 1 |
| minOccurs | Optional. Specifies the minimum number of times the any element can occur in the parent element. The value can be any number >= 0. Default value is 1 |
| namespace | Optional. Specifies the namespaces containing the elements that can be used. Can be set to one of the following:   * ##any - elements from any namespace is allowed (this is default) * ##other - elements from any namespace that is not the namespace of the parent element can be present * ##local - elements must come from no namespace * ##targetNamespace - elements from the namespace of the parent element can be present * List of {URI references of namespaces, ##targetNamespace, ##local} - elements from a space-delimited list of the namespaces can be present |
| processContents | Optional. Specifies how the XML processor should handle validation against the elements specified by this any element. Can be set to one of the following:   * strict - the XML processor must obtain the schema for the required namespaces and validate the elements (this is default) * lax - same as strict but; if the schema cannot be obtained, no errors will occur * skip - The XML processor does not attempt to validate any elements from the specified namespaces |
| *any attributes* | Optional. Specifies any other attributes with non-schema namespace |

### **Example 1**

The following example shows a declaration for an element called "person". By using the <any> element the author can extend (after <lastname>) the content of "person" with any element:

<xs:element name="person">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="firstname" type="xs:string"/>  
      <xs:element name="lastname" type="xs:string"/>  
      <xs:any minOccurs="0"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>