The Java™ WSIT Tutorial

For Web Services Interoperability Technologies
Milestone Release 6

August 24, 2007

Contents

About This Tutorial

THIS tutorial explains how to develop web applications using the Web Service Interoperability Technologies (WSIT). The tutorial describes how, when, and why to use the WSIT technologies and also describes the features and options that each technology supports.

WSIT, developed by Sun Microsystems, implements several new web services technologies including WS-Security, WS-Trust, WS-SecureConversation, WS-ReliableMessaging, WS-AtomicTransactions, Data Binding, and Optimization. WSIT was also tested in a joint effort by Sun Microsystems, Inc. and Microsoft with the expressed goal of ensuring interoperability between web services applications developed using either WSIT and the Windows Communication Foundation (WCF) product.

Who Should Use This Tutorial

This tutorial is intended for programmers who are interested in developing and deploying Java based clients and service providers that can interoperate with Microsoft .NET 3.0 clients and service providers.

How to Use This Tutorial

This tutorial addresses the following technology areas:

- Bootstrapping and Configuration
- Message Optimization
- Reliable Messaging (WS-RM)
- Web Services Security 1.1 (WS-Security)
- Web Services Trust (WS-Trust)
- Web Services Secure Conversation (WS-Secure Conversation)
- Data Contracts
- Atomic Transactions (WS-AT)

About the Examples

This section tells you everything you need to know to install, build, and run the examples.

Required Software

To use this tutorial you must download and install the following software:

- The latest Java SE 5.0 (Update 12) or JDK 6.0 (Update 2) with which the WSIT Milestone 6 software has been extensively tested
- GlassFish version 2 Build 58, your web container

  You can run the examples in this tutorial that use a web container *without* the NetBeans IDE on either GlassFish or Tomcat. However, for this edition of the tutorial, you can only run the examples that use a web container and the NetBeans IDE with GlassFish.
- WSIT distribution Milestone Release 6 (Release Candidate 1)
- Netbeans IDE 5.5.1 FCS
- WSIT plug-in modules, Version 2.41, for Netbeans IDE 5.5.1

See the *WSIT Installation Instructions*, located at `https://wsit-docs.dev.java.net/releases/m6/install.html`, for instructions about downloading and installing all the required software.

To run the examples described in this tutorial, you must also download the WSIT samples kits. Download the sample kits from the following locations:

- `https://wsit.dev.java.net/source/browse/*checkout*/wsit/wsit/docs/howto/wsit-enabled-fromjava.zip`
- `https://wsit.dev.java.net/source/browse/*checkout*/wsit/wsit/docs/howto/wsit-enabled-fromwsdl.zip`
- `https://wsit.dev.java.net/source/browse/*checkout*/wsit/wsit/docs/howto/csclient-enabled-fromjava.zip`
- `https://wsit-docs.dev.java.net/releases/m6/wsittutorial.zip`

Typographical Conventions

Table 1 lists the typographical conventions used in this tutorial.

Table 1 Typographical Conventions

Font Style	Uses
italic	Emphasis, titles, first occurrence of terms
`monospace`	URLs, code examples, file names, path names, tool names, application names, programming language keywords, tag, interface, class, method, field names, and properties
`italic monospace`	Variables in code, file paths, and URLs
<*`italic monospace`*>	User-selected file path components

Menu selections indicated with the right-arrow character →, for example, First→Second, should be interpreted as: select the First menu, then choose Second from the First submenu.

Feedback

Please send comments, broken link reports, errors, suggestions, and questions about this tutorial to the tutorial team at `users@wsit.dev.java.net`.

1 Introduction

This tutorial describes how to use the Web Services Interoperability Technologies (WSIT)—a product of Sun Microsystems web services interoperability effort to develop Java clients and service providers that interoperate with Microsoft .NET 3.0 clients and service providers.

The tutorial consists of the following chapters:

- This chapter, the introduction, introduces WSIT, highlights the features of each WSIT technology, describes the standards that WSIT implements for each technology, and provides high-level descriptions of how each technology works.
- Chapter 2 provides instructions for creating, deploying, and testing Web service providers and clients using NetBeans IDE.
- Chapter 3 describes how to create a WSIT client from a Web Service Description Language (WSDL) file.
- Chapter 4 describes how to configure web service providers and clients to use message optimization.
- Chapter 5 describes how to configure web service providers and clients to use reliable messaging.
- Chapter 6 describes how to use the NetBeans IDE to configure web service providers and clients to use web services security.
- Chapter 7 provides code examples and instructions for creating, deploying, and testing web service providers and clients using either of the supported web containers.

- Chapter 8 describes how to build and run a Microsoft Windows Communication Foundation (WCF) client that accesses the `addnumbers` service described in Chapter 7.
- Chapter 9 describes the best practices for production and consumption of *data contracts* for interoperability between WCF web services and Java web service clients or Java web services and WCF web service clients.
- Chapter 10 describes Atomic Transactions.

What is WSIT?

Sun is working closely with Microsoft to ensure interoperability of web services enterprise technologies such as message optimization, reliable messaging, and security. The initial release of WSIT is a product of this joint effort. WSIT is an implementation of a number of open web services specifications to support enterprise features. In addition to message optimization, reliable messaging, and security, WSIT includes a bootstrapping and configuration technology. Figure 1–1 shows the underlying services that were implemented for each technology.

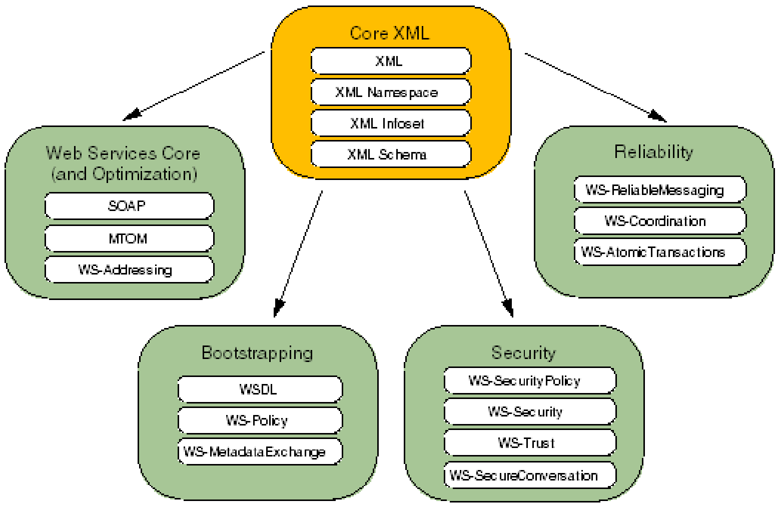


Figure 1–1 WSIT Web Services Features

Starting with the core XML support currently built into the Java platform, WSIT uses or extends existing features and adds new support for interoperable web services. See the following sections for an overview of each feature:

- Bootstrapping and Configuration (page 3)
- Message Optimization Technology (page 4)
- Reliable Messaging Technology (page 5)
- Security Technology (page 6)

Bootstrapping and Configuration

Bootstrapping and configuration consists of using a URL to access a web service, retrieving its WSDL file, and using the WSDL file to create a web service client that can access and consume a web service. The process consists of the following steps, which are shown in Figure 1–2:

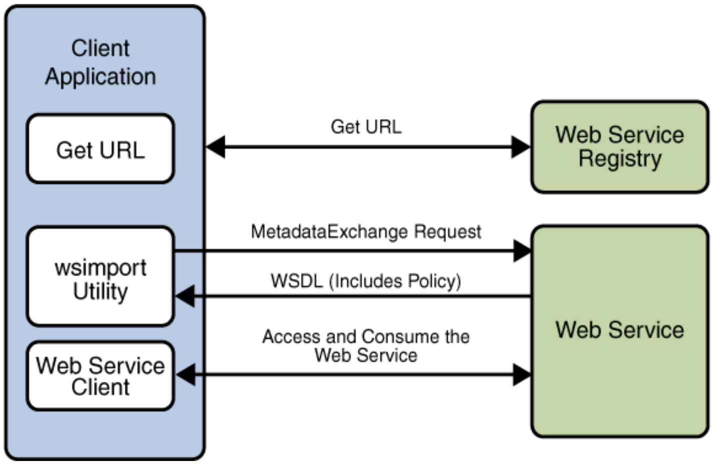


Figure 1–2 Bootstrapping and Configuration

1. Client acquires the URL for a web service that it wants to access and consume. How you acquire the URL is outside the scope of this tutorial. For example, you might look up the URL in a Web Services registry.
2. The client uses the URL and the `wsimport` tool to send a MetadataExchangeRequest to access the web service and retrieve the WSDL file. The WSDL file contains a description of the web service endpoint, including WS-Policy assertions that describe the security and/or reliability capabili-

ties and requirements of the service. The description describes the requirements that must be satisfied to access and consume the web service.

3. The client uses the WSDL file to create the web service client.
4. The web service client accesses and consumes the web service.

Chapter 3 explains how to bootstrap and configure a web service client and a web service endpoint that use the WSIT technologies.

Message Optimization Technology

A primary function of web services applications is to share data among applications over the Internet. The data shared can vary in format and include large binary payloads, such as documents, images, music files, and so on. When large binary objects are encoded into XML format for inclusion in SOAP messages, even larger files are produced. When a web service processes and transmits these large files over the network, the performance of the web service application and the network are negatively affected. In the worst case scenario the effects are as follows:

- The performance of the web service application degrades to a point that it is no longer useful.
- The network gets bogged down with more traffic than the allotted bandwidth can handle.

One way to deal with this problem is to encode the binary objects so as to optimize both the SOAP application processing time and the bandwidth required to transmit the SOAP message over the network. In short, XML needs to be optimized for web services. This is the exactly what the Message Optimization technology does. It ensures that web services messages are transmitted over the Internet in the most efficient manner.

Sun recommends that you use message optimization if your web service client or web service endpoint will be required to process binary encoded XML documents larger than 1KB.

For instructions on how to use the Message Optimization technology, see Chapter 4.

Reliable Messaging Technology

Reliable Messaging is a Quality of Service (QoS) technology for building more reliable web services. Reliability is measured by a system’s ability to deliver messages from point A to point B without error. The primary purpose of Reliable Messaging is to ensure the delivery of application messages to web service endpoints.

The reliable messaging technology ensures that messages in a given message sequence are delivered at least once and not more than once and optionally in the correct order. When messages in a given sequence are lost in transit or delivered out of order, this technology enables systems to recover from such failures. If a message is lost in transit, the sending system retransmits the message until its receipt is acknowledged by the receiving system. If messages are received out of order, the receiving system may re-order the messages into the correct order.

The Reliable Messaging technology can also be used to implement session management. A unique message sequence is created for each client-side proxy and the lifetime of the sequence identifier coincides with the lifetime of the proxy. Therefore, each message sequence can be viewed as a session and can be used to implement session management.

You should consider using reliable messaging if the web service is experiencing the following types of problems:

- Communication failures are occurring that result in the network being unavailable or connections being dropped
- Application messages are being lost in transit
- Application messages are arriving at their destination out of order and ordered delivery is a requirement

To help decide whether or not to use reliable messaging, weigh the following advantages and disadvantages:

- Enabling reliable messaging ensures that messages are delivered exactly once from the source to the destination and, if the ordered-delivery option is enabled, ensures that messages are delivered in order.
- Enabling reliable messaging causes a degradation of web service performance, especially if the ordered delivery option is enabled.
- Non-reliable messaging clients cannot interoperate with web services that have reliable messaging enabled.

For instructions on how to use the Reliable Messaging technology, see Chapter 5.

Security Technology

Until now, web services have relied on transport-based security such as SSL to provide point-to-point security. WSIT implements WS-Security so as to provide interoperable message content integrity and confidentiality, even when messages pass through intermediary nodes before reaching their destination endpoint. WS-Security as provided by WSIT is in addition to existing transport-level security, which may still be used.

WSIT also enhances security by implementing WS-Secure Conversation, which enables a consumer and provider to establish a shared security context when a multiple-message-exchange sequence is first initiated. Subsequent messages use derived session keys that increase the overall security while reducing the security processing overhead for each message.

Further, WSIT implements two additional features to improve security in web services:

- Web Services Security Policy—Enables web services to use security assertions to clearly represent security preferences and requirements for web service endpoints.
- Web Services Trust—Enables web service applications to use SOAP messages to request security tokens that can then be used to establish trusted communications between a client and a web service.

WSIT implements these features in such a way as to ensure that web service binding security requirements, as defined in the WSDL file, can interoperate with and be consumed by WSIT and WCF endpoints.

For instructions on how to use the WS-Security technology, see Chapter 6.

How WSIT Relates to Windows Communication Foundation (WCF)

Web services interoperability is an initiative of Sun and Microsoft. The goal is to produce web services consumers and producers that support platform independence, and then to test and deliver products to market that interoperate across different platforms.

WSIT is the product of Sun's web services interoperability initiative. Windows Communication Foundation (WCF) is Microsoft's unified programming model for building connected systems. WCF, which is now available as part of the

.NET Framework 3.0 product, includes application programming interfaces (APIs) for building secure, reliable, transacted web services that interoperate with non-Microsoft platforms.

In a joint effort, Sun Microsystems and Microsoft are testing WSIT against WCF to ensure that Sun web service clients (consumers) and web services (producers) do in fact interoperate with WCF web services applications and vice versa. The testing will ensure that the following interoperability goals are realized:

- WSIT web services clients can access and consume WCF web services.
- WCF web services clients can access and consume WSIT web services.

Sun is building WSIT on the Java platform and Microsoft is building WCF on the .NET 3.0 platform. The sections that follow describe the web services specifications implemented by Sun Microsystems in Web Services Interoperability Technologies (WSIT) and provide high-level descriptions of how each WSIT technology works.

Note: Because WSIT-based clients and services are interoperable, you can gain the benefits of WSIT without using WCF.

WSIT Specifications

The specifications for bootstrapping and configuration, message optimization, reliable messaging, and security technologies are discussed in the following sections:

- Bootstrapping and Configuration Specifications (page 8)
- Message Optimization Specifications (page 10)
- Reliable Messaging Specifications (page 12)
- Security Specifications (page 13)

WSIT 1.0 implements the following versions of these specifications:

- Bootstrapping
    - WS-MetadataExchange v1.1
- Reliable Messaging
    - WS-ReliableMessaging v1.0

    - WS-ReliableMessaging Policy v1.0
- Atomic Transactions
    - WS-AtomicTransaction v1.0
    - WS-Coordination v1.0
- Security
    - WS-Security v1.1
    - WS-SecurityPolicy v1.1
    - WS-Trust v1.0
    - WS-SecureConversation v1.0
- Policy
    - WS-Policy v1.2
    - WS-PolicyAttachment v1.2

The same versions of these specifications are also implemented in WCF in .NET 3.0. Sun will update to the standard versions of these specifications in a future release of WSIT. Those versions will coincide with the versions used in WCF in .NET 3.5.

Bootstrapping and Configuration Specifications

Bootstrapping and configuring involves a client getting a web service URL (perhaps via service registry) and obtaining the information needed to build a web services client that is capable of accessing and consuming a web service over the Internet. This information is usually obtained from a WSDL file. Figure 1–2 shows the specifications that were implemented to support bootstrapping and configuration.

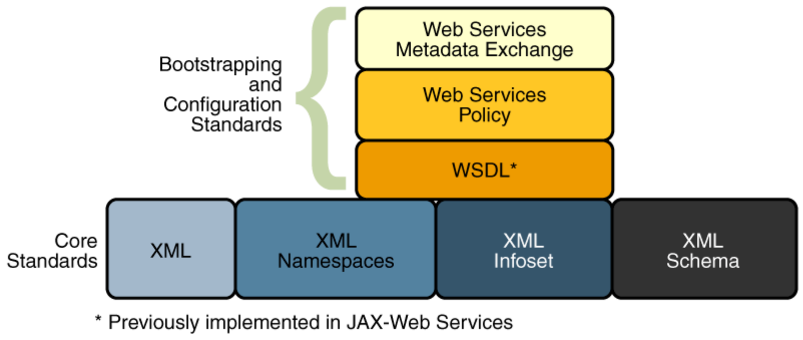

Bootstrapping and Configuration Standards
Web Services Metadata Exchange
Web Services Policy
WSDL*
Core Standards
XML
XML Namespaces
XML Infoset
XML Schema
* Previously implemented in JAX-Web Services

Figure 1–3 Bootstrapping and Configuration Specifications

In addition to the Core XML specifications, bootstrapping and configuration was implemented using the following specifications:

- **WSDL**: The Web Services Description Language (WSDL) specification was previously implemented in JAX-WS. WSDL is a standardized XML format for describing network services. The description includes the name of the service, the location of the service, and ways to communicate with the service, that is, what transport to use. WSDL descriptions can be stored in service registries, published on the Internet, or both.
- **Web Services Policy**: This specification provides a flexible and extensible grammar for expressing the capabilities, requirements, and general characteristics of a web service. It provides the mechanisms needed to enable web services applications to specify policy information in a standardized way. However, this specification does not provide a protocol that constitutes a negotiation or message exchange solution for web Services. Rather, it specifies a building block that is used in conjunction with the WS-Metadata Exchange protocol. When applied in the web services model, policy is used to convey conditions on interactions between two web service endpoints. Typically, the provider of a web service exposes a policy to convey conditions under which it provides the service. A requester might use the policy to decide whether or not to use the service.
- **Web Services Metadata Exchange**: This specification defines a protocol to enable a consumer to obtain a web service’s metadata, that is, its WSDL and policies. It can be thought of as a bootstrap mechanism for communication.

Message Optimization Specifications

Message optimization is the process of transmitting web services messages in the most efficient manner. It is achieved in web services communication by encoding messages prior to transmission and then de-encoding them when they reach their final destination.

Figure 1–4 shows the specifications that were implemented to optimize communication between two web service endpoints.

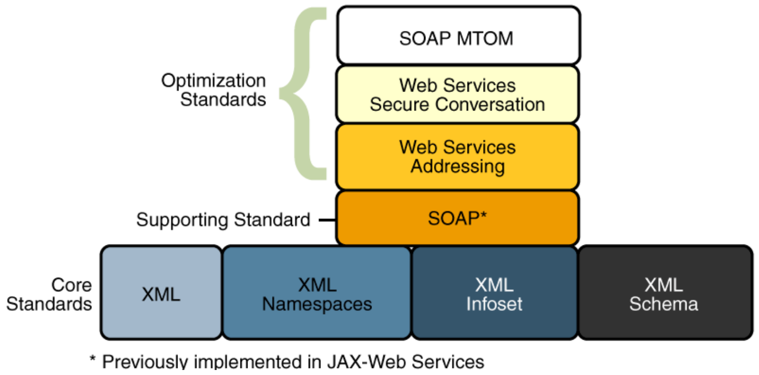


Figure 1–4 Message Optimization Specifications

In addition to the Core XML specifications, optimization was implemented using the following specifications:

- **SOAP**: JAX Web Services currently supports the SOAP wire protocol. With SOAP implementations, client requests and web service responses are most often transmitted as Simple Object Access Protocol (SOAP) messages over HTTP to enable a completely interoperable exchange between clients and web services, all running on different platforms and at various locations on the Internet. HTTP is a familiar request-and response standard for sending messages over the Internet, and SOAP is an XML-based protocol that follows the HTTP request-and-response model. In SOAP 1.1, the SOAP portion of a transported message handles the following:
  - Defines an XML-based envelope to describe what is in the message and how to process the message.
  - Includes XML-based encoding rules to express instances of application-defined data types within the message.
  - Defines an XML-based convention for representing the request to the remote service and the resulting response.

In SOAP 1.2 implementations, web service endpoint addresses can be included in the XML-based SOAP envelope, rather than in the transport header (for example in the HTTP transport header), thus enabling SOAP messages to be transport independent.

- **Web Services Addressing**: The Java APIs for W3C Web Services Addressing were first shipped with Java Web Services Developer's Pack 2.0 (JWSDP 2.0). This specification defines a set of abstract properties and an XML Infoset representation that can be bound to a SOAP message so as to reference web services and to facilitate end-to-end addressing of endpoints in messages. A web service endpoint is an entity, processor, or resource that can be referenced and to which web services messages can be addressed. Endpoint references convey the information needed to address a web service endpoint. The specification defines two constructs: message addressing properties and endpoint references, that normalize the information typically provided by transport protocols and messaging systems in a way that is independent of any particular transport or messaging system. This is accomplished by defining XML tags for including web service addresses in the SOAP message, instead of the HTTP header. The implementation of these features enables messaging systems to support message transmission—in a transport-neutral manner—through networks that include processing nodes such as endpoint managers, firewalls, and gateways.
- **Web Services Secure Conversation**: This specification provides better message-level security and efficiency in multiple-message exchanges in a standardized way. It defines basic mechanisms on top of which secure messaging semantics can be defined for multiple-message exchanges and allows for contexts to be established and potentially more efficient keys or new key material to be exchanged, thereby increasing the overall performance and security of the subsequent exchanges.
- **SOAP MTOM**: The SOAP Message Transmission Optimization Mechanism (MTOM), paired with the XML-binary Optimized Packaging (XOP), provides standard mechanisms for optimizing the transmission and/or wire format of SOAP messages by selectively encoding portions of the SOAP message, while still presenting an XML Infoset to the SOAP application. This mechanism enables the definition of a hop-by-hop contract between a SOAP node and the next SOAP node in the SOAP message path so as to facilitate the efficient pass-through of optimized data contained within headers or bodies of SOAP messages that are relayed by an intermediary. Further, it enables message optimization to be done in a binding independent way.

Reliable Messaging Specifications

Reliability is measured by a system's ability to deliver messages from point A to point B without error. Figure 1–5 shows the specifications that were implemented to ensure reliable delivery of messages between two web services endpoints.

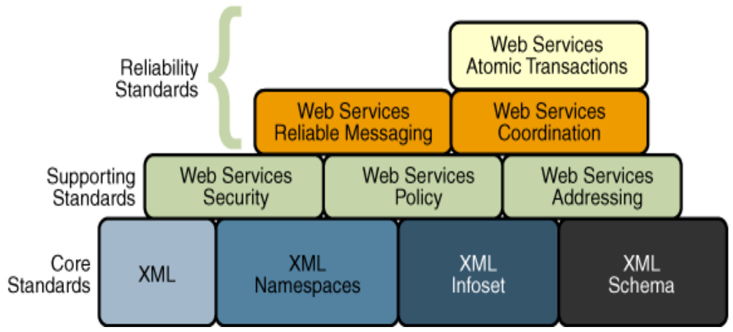


Figure 1–5 Reliable Messaging Specifications

In addition to the Core XML specifications and supporting standards (Web Services Security and Web Services Policy—which are required building blocks), the reliability feature is implemented using the following specifications:

- **Web Services Reliable Messaging**: This specification defines a standardized way to identify, track, and manage the reliable delivery of messages between exactly two parties, a source and a destination, so as to recover from failures caused by messages being lost or received out of order. The specification is also extensible so it allows additional functionality, such as security, to be tightly integrated. The implementation of this specification integrates with and complements the Web Services Security, and the Web Services Policy implementations.
- **Web Services Coordination**: This specification defines a framework for providing protocols that coordinate the actions of distributed applications. This framework is used by Web Services Atomic Transactions. The implementation of this specification enables the following capabilities:
  - Enables an application service to create the context needed to propagate an activity to other services and to register for coordination protocols.
  - Enables existing transaction processing, workflow, and other coordination systems to hide their proprietary protocols and to operate in a heterogeneous environment.

   - Defines the structure of context and the requirements so that context can be propagated between cooperating services.
- **Web Services Atomic Transactions**: This specification defines a standardized way to support two-phase commit semantics such that either all operations invoked within an atomic transaction succeed or are all rolled back. Implementations of this specification require the implementation of the Web Services Coordination specification.

Security Specifications

Figure 1–6 shows the specifications implemented to secure communication between two web service endpoints and across intermediate endpoints.

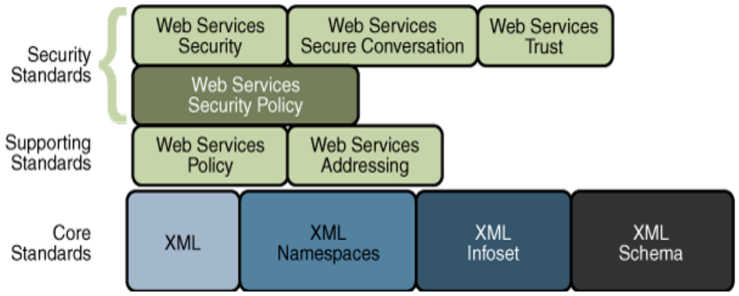


Figure 1–6 Web Services Security Specifications

In addition to the Core XML specifications, the security feature is implemented using the following specifications:

- **Web Services Security**: This specification defines a standard set of SOAP extensions that can be used when building secure web services to implement message content integrity and confidentiality. The implementation provides message content integrity and confidentiality even when communication traverses intermediate nodes, thus overcoming a short coming of SSL. The implementation can be used within a wide variety of security models including PKI, Kerberos, and SSL and provides support for multiple security token formats, multiple trust domains, multiple signature formats, and multiple encryption technologies.
- **Web Services Policy**: This specification provides a flexible and extensible grammar for expressing the capabilities, requirements, and general characteristics of a web service. It provides a framework and a model for the

expression of these properties as policies and is a building block for Web Services Security policy.

- **Web Services Trust**: This specification supports the following capabilities in a standardized way:
  - Defines extensions to Web Services Security that provide methods for issuing, renewing, and validating security tokens used by Web services security.
  - Establishes, assesses the presence of, and brokers trust relationships.
- **Web Services Secure Conversation**: This specification defines a standardized way to provide better message-level security and efficiency in multiple-message exchanges. The WSIT implementation provides basic mechanisms on top of which secure messaging semantics can be defined for multiple-message exchanges and allows for contexts to be established along with more efficient keys or new key material. This approach increases the overall performance and security of the subsequent exchanges. While the Web Services Security specification, described above, focuses on the message authentication model, it does leave openings for several forms of attacks. The Secure Conversation authentication specification defines a standardized way to authenticate a series of messages, thereby addressing the short comings of Web Services Security. With the Web Services Security Conversation model, the security context is defined as a new Web Services security token type that is obtained using a binding of Web Services Trust.
- **Web Services Security Policy**: This specification defines a standard set of patterns or sets of assertions that represent common ways to describe how messages are secured on a communications path. The WSIT implementation allows flexibility in terms of tokens, cryptography, and mechanisms used, including leveraging transport security, but is specific enough to ensure interoperability based on assertion matching by web service clients and web services providers.

How the WSIT Technologies Work

The following sections provide a high-level description of how the messaage optimization, reliable messaging, and security technologies work.

How Message Optimization Works

Message optimization ensures that web services messages are transmitted over the Internet in the most efficient manner. Because XML is a textual format, binary files must be represented using character sequences before they can be embedded in an XML document. A popular encoding that permits this embedding is known as base64 encoding, which corresponds to the XML Schema data type xsd:base64Binary. In a web services toolkit that supports a binding framework, a value of this type must be encoded before transmission and decoded before binding. The encoding and decoding process is expensive and the costs increase linearly as the size of the binary object increases.

Message optimization enables web service endpoints to identify large binary message payloads, remove the message payloads from the body of the SOAP message, encode the message payloads using an efficient encoding mechanism (effectively reducing the size of the payloads), re-insert the message payloads into the SOAP message as attachments (the file is linked to the SOAP message body by means of an Include tag). Thus, message optimization is achieved by encoding binary objects prior to transmission and then de-encoding them when they reach there final destination.

The optimization process is really quite simple. To effect optimized message transmissions, the sending endpoint checks the body of the SOAP message for XML encoded binary objects that exceed a predetermined size and encodes those objects for efficient transmission over the Internet.

SOAP MTOM, paired with the XML-binary Optimized Packaging (XOP), addresses the inefficiencies related to the transmission of binary data in SOAP documents. Using MTOM and XOP, XML messages are dissected in order to transmit binary files as MIME attachments in a way that is transparent to the application. This transformation is restricted to base64 content in canonical form as defined in XSD Datatypes as specified in *XML Schema Part 2: Datatypes Second Edition, W3C Recommendation 28 October 2004*.

Thus, the WSIT technology achieves message optimization through an implementation of the MTOM and XOP specifications. With the message optimization feature enabled, small binary objects are sent in-line in the SOAP body. For large binary objects, this becomes quite inefficient, so the binary object is separated from the SOAP body, encoded, sent as an attachment to the SOAP message, and decoded when it reaches its destination endpoint.

How Reliable Messaging Works

When reliable messaging is enabled, messages are grouped into sequences, which are defined by the client’s proxies. Each proxy corresponds to a message sequence, which consists of all of the request messages for that proxy. Each message contains a sequence header. The header includes a sequence identifier that identifies the sequence and a unique message number that indicates the order of the message in the sequence. The web service endpoint uses the sequence header information to group the messages and—if the Ordered Delivery option is selected—to process them in the proper order. Additionally, if secure conversation is enabled, each message sequence is assigned its own security context token. The security context token is used to sign the handshake messages that initialize communication between two web service endpoints and subsequent application messages.

Thus, using the Reliable Messaging technology, web service endpoints collaborate to determine which messages in a particular application message sequence arrived at the destination endpoint and which messages require resending. The reliable messaging protocol requires that the destination endpoint return message-receipt acknowledgements that include the sequence identifier and the message number of each message received. If the source determines that a message was not received by the destination, it resends the message and requests an acknowledgement. Once the source has sent all messages for a given sequence and their receipt has been acknowledged by the destination, the source terminates the sequence.

The web service destination endpoint in turn sends the application messages along to the application. If ordered delivery is configured (optional), the destination endpoint reconstructs a complete stream of messages for each sequence in the exact order in which the messages were sent and sends them along to the destination application. Thus, through the use of the reliable messaging protocol, the destination endpoint is able to provide the following "delivery assurances" to the web service application:

- Each message is delivered to the destination application at least once.
- Each message is delivered to the destination application at most once.
- Sequences of messages are grouped by sequence identifiers and delivered to the destination application in the order defined by the message numbers.

Figure 1–7 shows a simplified view of client and web service application message exchanges when the Reliable Messaging protocol is not used.

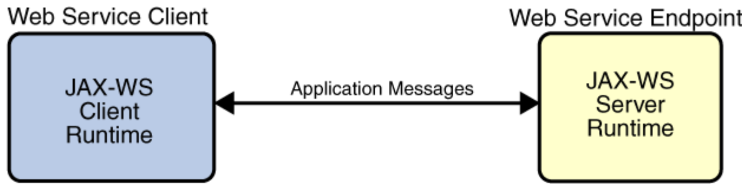


Figure 1–7 Application Message Exchange Without Reliable Messaging

When the Reliable Messaging protocol is not used, application messages flow over the HTTP connection with no delivery assurances. If messages are lost in transit or delivered out of order, the communicating endpoints have no way of knowing.

Figure 1–8 shows a simplified view of client and web service application message exchanges when reliable messaging is enabled.

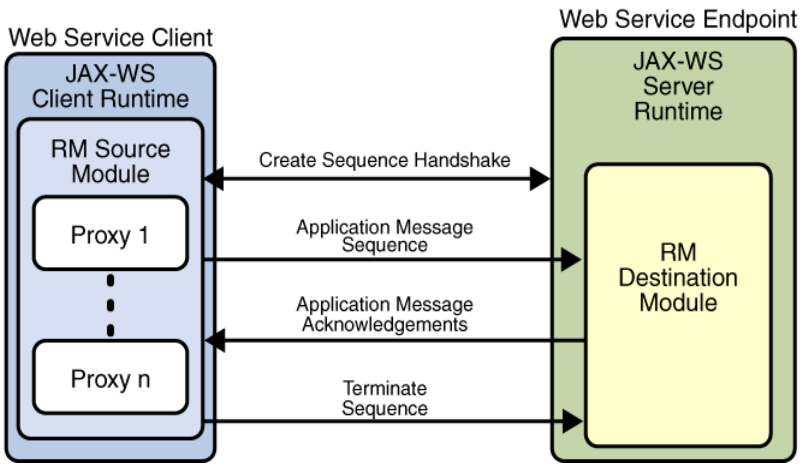


Figure 1–8 Application Message Exchange with Reliable Messaging Enabled

With reliable messaging enabled, the Reliable Messaging source module is plugged into the JAX-WS web service client. The source module transmits the application messages and keeps copies of the messages until their receipt is acknowledged by the destination module via the exchange of protocol messages. The destination module acknowledges messages and optionally buffers them for ordered-delivery guarantee. After guaranteeing order, if configured, the destination module allows the messages to proceed through the JAX-WS dispatch for delivery to the endpoint or application destination.

How Security Works

The following sections describe how the WSIT security technologies, security policy, trust, and secure conversation work.

How Security Policy Works

The WSIT Web Service Security Policy implementation builds on the features provided by the Web Service Policy implementation in WSIT. It enables users to use XML elements to specify the security requirements of a web service endpoint, that is, how messages are secured on the communication path between the client and the web service. The web service endpoint specifies the security requirements to the client as assertions (see Figure 1–9).

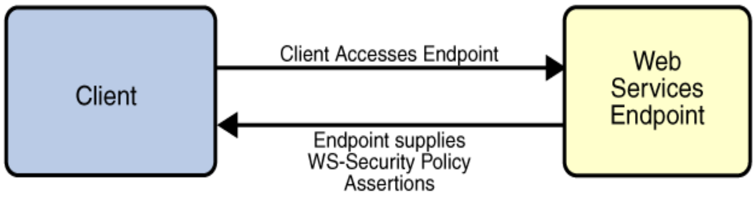


Figure 1–9 Security Policy Exchange

The security policy model uses the policy specified in the WSDL file for associating policy assertions with web service communication. As a result, whenever possible, the security policy assertions do not use parameters or attributes. This enables first-level, QName-based assertion matching to be done at the framework level without security domain-specific knowledge. The first-level matching provides a narrowed set of policy alternatives that are shared by the client and web service endpoint when they attempt to establish a secure communication path.

Note: A QName is a qualified name, as specified by XML Schema Part2: Datatypes specification, Namespaces in XML, Namespaces in XML Errata. A qualified name is made up of a namespace URI, a local part, and a prefix.

The benefit of representing security requirements as assertions is that QName matching is sufficient to find common security alternatives and that many aspects of security can be factored out and re-used. For example, it may be common that the security mechanism is constant for a web service endpoint, but that the message parts that are protected, or secured, may vary by message action.

The following types of assertions are supported:

- **Protection assertions**: Define the scope of security protection. These assertions identify the message parts that are to be protected and how, that is, whether data integrity and confidentiality mechanisms are to be used.
- **Conditional assertions**: Define general aspects or pre-conditions of the security. These assertions define the relationships within and the characteristics of the environment in which security is being applied, such as the tokens that can be used, which tokens are for integrity or confidentiality protection, and applicable algorithms to use, and so on.
- **Security binding assertions**: Define the security mechanism that is used to provide security. These assertions are logical grouping that defines how the conditional assertions are used to protect the indicated message parts. For example, that an asymmetric token is to be used with a digital signature to provide integrity protection, and that parts are to be encrypted with a symmetric key, which is then encrypted using the public key of the recipient. In its simplest form, the security binding assertions restrict what can be placed in the `wsse:Security` header and the associated processing rules.
- **Supporting token assertions**: Define the token types and usage patterns that can be used to secure individual operations and/or parts of messages.
- **Web Services Security and Trust assertions**: Define the token referencing and trust options that can be used.

How Trust Works

Figure 1–11 shows how the Web Services Trust technology establishes trust.

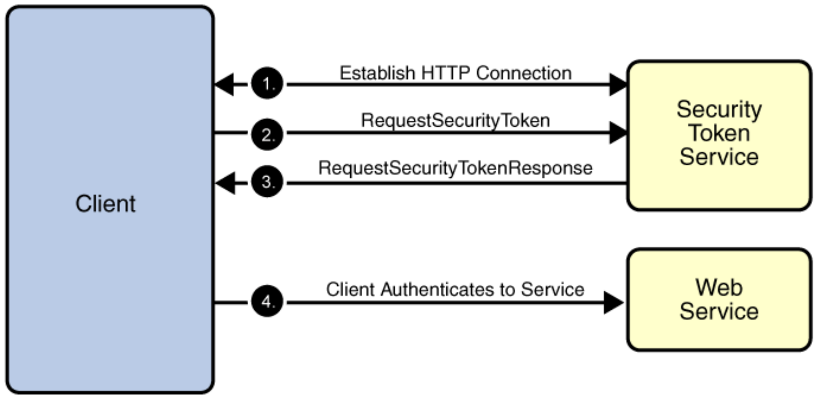

Client
1.
Establish HTTP Connection
2.
RequestSecurityToken
3.
RequestSecurityTokenResponse
Security Token Service
4.
Client Authenticates to Service
Web Service

Figure 1–10 Trust and Secure Conversation

To establish trust between a client, a Security Token Service, and a web service:

1. The client establishes an HTTPS connection with the Secure Token Service using one of the following methods:
   - **Username Authentication and Transport Security**: The client authenticates to the Security Token Service using a username token. The Security Token Service uses a certificate to authenticate to the Client. Transport security is used for message protection.
   - **Mutual Authentication**: Both the client-side and server-side use X509 certificates to authenticate to each other. The client request is signed using Client's X509 certificate, then signed using ephemeral key. The web service signs the response using keys derived from the client's key.
2. The client sends a RequestSecurityToken message to the Security Token Service.
3. The Security Token Service sends a Security Assertion Markup Language (SAML) token to the Client.
4. The client uses the SAML token to authenticate itself to the web service and trust is established.

All communication uses SOAP messages.

How Secure Conversation Works

Figure 1–11 shows how the Web Services Secure Conversation technology establishes a secure conversation when the Trust technology is not used.

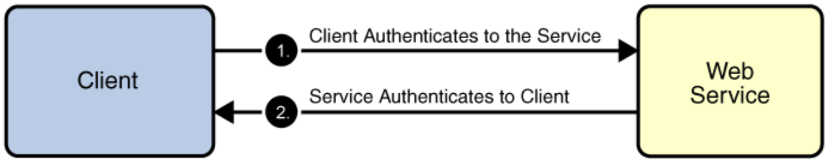

Client
1.
Client Authenticates to the Service
2.
Service Authenticates to Client
Web Service

Figure 1–11 Secure Conversation

To establish a secure conversation between a Client and a web service:

1. The client sends a X509 Certificate to authenticate itself to the web service.
2. The web service sends a X509 Certificate to authenticate itself to the client.

All communication uses SOAP messages.

2 WSIT Example Using a Web Container and NetBeans

THIS chapter describes how to use NetBeans IDE and GlassFish to build and deploy a web service and client that use WSIT technologies. It includes examples of the files that the IDE helps you create and examples of the build directories and the key files that the IDE produces to create a web service and a client.

This chapter covers the following topics:

Registering GlassFish with the IDE

Before you create the web service, perform the following steps to register Glassfish with the IDE:

1. Start the IDE and choose Tools→Server Manager from the main window. The Server Manager window appears.
2. Click Add Server. Select the Sun Java System Application Server, assign a name to server instance, and click Next. The platform folder location window appears.
3. Specify the platform location of the server instance, and the domain to which you want to register, and click Finish. The Server Manager window appears.
4. Enter the server username and password that you supplied when you installer the web container (the default is `admin/adminadmin`) and click Close.

Creating a Web Service

The starting point for developing a web service to use the WSIT technologies is a Java class file annotated with the `javax.jws.WebService` annotation. The `WebService` annotation defines the class as a web service endpoint. The following Java code shows a web service. The IDE will create most of this Java code for you.

```
package org.me.calculator;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

@WebService()
public class Calculator {
  @WebMethod(action="sample_operation")
  public String operation(@WebParam(name="param_name")
       String param) {
    // implement the web service operation here
    return param;
  }

  @WebMethod(action=”add”)
  public int add(@WebParam(name = "i") int i,
       @WebParam(name = "j") int j) {
    int k = i + j;
    return k;
  }
}
```

Notice that this web service performs a very simple operation. It takes two integers, adds them, and returns the result.

Perform the following steps to use the IDE to create this web service:

1. Click on the Runtime tab in the left pane and verify that GlassFish is listed in the left pane. If it is not listed, refer to Registering GlassFish with the IDE (page 23) and register it.
2. Choose File→New Project, select Web Application from the Web category, and click Next.
3. Assign the project a name that is representative of services that will be provided by the web service (for example, CalculatorApplication), set the Project Location to the location of the Sun application server, and click Finish.

Note: As of this writing, when creating the web service project be sure to define a Project Location that does not include spaces in the directory name. Spaces in the directory might cause the web service and web service clients to fail to build and deploy properly. To avoid this problem, Sun recommends that you create a directory, for example `C:\work`, and put your project there.

---

4. Right-click the CalculatorApplication node and choose New→Web Service.
5. Enter the web service name (`CalculatorWS`) and the package name (`org.me.calculator`) in the Web Service Name and the Package fields respectively.
6. Select Create an Empty Web Service.
7. Click Finish.

   The IDE then creates a skeleton `CalculatorWS.java` file for the web service that includes an empty `WebService` class with annotation `@Webservice`.

8. Right-click within the body of the class and choose Web Service→Add Operation.
9. In the upper part of the Add Operation dialog box, type `add` in Name and choose `int` from the Return Type drop-down list.
10. In the lower part of the Add Operation dialog box, click Add and create a parameter of type `int` named `i`. Click OK. Click Add again and create a parameter of type `int` called `j`. Click OK and close the Enter Method Parameter dialog box.

11. Click OK at the bottom of the Add Operation dialog box.
12. Notice that the `add` method has been added to the Source Editor:

```
@WebMethod
public int add(@WebParam(name = "i") int i,
     @WebParam(name = "j") int j) {
   // TODO implement operation
   return 0;
}
```

13. Change the add method to the following (changes are in bold):

```
@WebMethod(action="add")
public int add(@WebParam(name = "i") int i,
     @WebParam(name = "j") int j) {
   int k = i + j;
   return k;
}
```

Note: To ensure interoperability with Windows Communication Foundation (WCF) clients, you must specify the `action` element of `@WebMethod` in your endpoint implementation classes. WCF clients will incorrectly generate an empty string for the Action header if you do not specify the `action` element.

14. Save the `CalculatorWS.java` file.

You have successfully coded the web service.

Configuring WSIT Features in the Web Service

Now that you have coded a web service, you can configure the web service to use WSIT technologies. This section only describes how to configure the WSIT Reliable Messaging technology. For a discussion of reliable messaging, see Chapter 5. To see how to secure the web service, see Chapter 6.

To configure a web service to use the WSIT Reliable Messaging technology, perform the following steps:

1. In the Projects window, expand the Web Services node under the CalculatorApplication node, right-click the CalculatorWS node, and choose Edit Web Service Attributes, as shown in Figure 2–1:

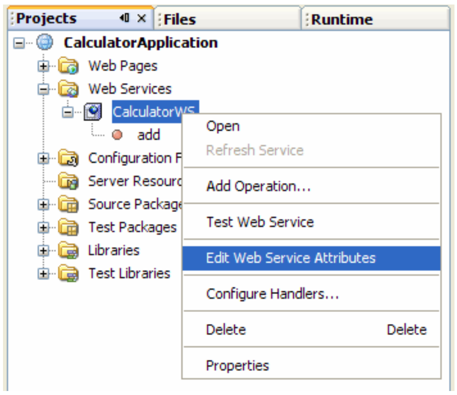
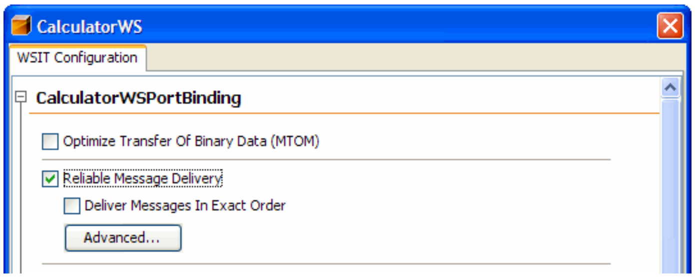


Figure 2–1 Accessing the Edit Web Service Attributes

The Web Service Attributes Editor appears.

2. Select the Reliable Message Delivery check box, as shown in Figure 2–2, and click OK.



☐ Optimize Transfer Of Binary Data (MTOM)
☑ Reliable Message Delivery
☐ Deliver Messages In Exact Order

Figure 2–2 Reliable Messaging Configuration Window

This setting ensures that the service sends an acknowledgement to the clients for each message that is delivered, thus enabling clients to recognize message delivery failures and to retransmit the message. This capability makes the web service a “reliable” web service.

3. In the left pane, expand the Web Pages node and the WEB-INF node, and open the `wsit-<`*`endpoint classname`*`>.xml` file in the Source Editor.

Notice that the IDE has added the following tags to the file to enable reliable messaging:

```
<wsp:Policy wsu:Id="CalculatorWSPortBindingPolicy">
   <wsp:ExactlyOne>
```

```
            <wsp:All>
               <wsaw:UsingAddressing xmlns:wsaws=
                  "http://www.w3.org/2006/05/addressing/wsdl"/>
               <wsrm:RMAssertion/>
            </wsp:All>
         </wsp:ExactlyOne>
      </wsp:Policy>
```

Deploying and Testing a Web Service

Now that you have configured the web service to use WSIT technologies, you can deploy and test it.

To deploy and test the web service, perform the following steps:

1. Right-click the project node, select Properties, and select Run.
2. Type `/CalculatorWSService?wsdl` in the Relative URL field and click OK.
3. Right-click the project node and choose Run Project. The first time Glassfish is started, you will be prompted for the admin password. The IDE starts the web container, builds the application, and displays the WSDL file page in your browser. You have now successfully tested the deployed a WSIT-enabled web service.

   Notice that the WSDL file includes the following WSIT tags:

   ```
   <wsp:UsingPolicy/>
   <wsp:Policy wsu:Id="CalculatorWSPortBindingPolicy">
      <wsp:ExactlyOne>
         <wsp:All>
            <ns1:RMAssertion/>
            <ns2:UsingAddressing/>
         </wsp:All>
      </wsp:ExactlyOne>
   </wsp:Policy>
   ```

You have now successfully tested the deployment of a WSIT-enabled web service.

Creating a Client to Consume a WSIT-Enabled Web Service

Now that you have built and tested a web service that uses WSIT technologies, you can create a client that accesses and consumes that web service. The client will use the web service's WSDL to create the functionality necessary to satisfy the interoperability requirements of the web service.

To create a client to access and consume the web service, perform the following steps:

1. Choose File→New Project, select Web Application from the Web category and click Next.
2. Name the project, for example, CalculatorWSServletClient, and click Finish.
3. Right-click the CalculatorWSServletClient node and select New→Web Service Client. The New Web Service Client window appears.

Note: NetBeans submenus are dynamic, so the Web Service Client option may not appear. If you do not see the Web Service Client option, select New→File\Folder→Webservices→Web Service Client.

---

4. Select the WSDL URL option.
5. Cut and paste the URL of the web service that you want the client to consume into the WSDL URL field. For example, here is the URL for the `CalculatorWS` web service:

```
http://localhost:8080/CalculatorApplication/CalculatorWSService?wsdl
```

   When JAX-WS generates the web service, it appends "Service" to the class name by default.

6. Type `org.me.calculator.client` in the Package field, and click Finish. The Projects window displays the new web service client, as shown in Figure 2–3.

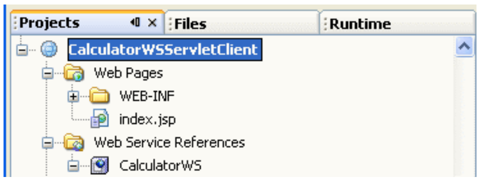


Figure 2–3 Web Service Client

7. Right-click the CalculatorWSServletClient project node and choose New→Servlet.
8. Name the servlet `ClientServlet`, specify the package name, for example, `org.me.calculator.client` and click Finish.
9. To make the servlet the entry point to your application, right-click the CalculatorWSServletClient project node, choose Properties, click Run, type `/ClientServlet` in the Relative URL field and click OK.
10. If `ClientServlet.java` is not already open in the Source Editor, open it.
11. In the Source Editor, remove the line that comments out the body of the `processRequest` method. This is the start-comment line that starts the section that comments out the code:

    ```
    /* TODO output your page here
    ```

12. Delete the end-comment line that ends the section of commented out code:

    ```
    */
    ```

13. Add some empty lines after the following line:

    ```
    out.println("<h1>Servlet ClientServlet at " +
        request.getContextPath () + "</h1>");
    ```

14. Right-click in one of the empty lines that you added. Choose Web Service Client Resources→Call Web Service Operation. The Select Operation to Invoke dialog box appears.
15. Browse to the Add operation and click OK. The `processRequest` method is as follows, with bold indicating code added by the IDE:

    ```
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Servlet ClientServlet</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>Servlet ClientServlet at " +
          request.getContextPath () + "</h1>");

    try { // Call Web Service Operation
      org.me.calculator.client.CalculatorWS port =
        service.getCalculatorWSPort();
      // TODO initialize WS operation arguments here
      int i = 0;
      int j = 0;
    ```

```
        // TODO process result here
        int result = port.add(i,  j);
        out.println("Result = "+result);
        }catch (Exception ex) {
        // TODO handle custom exceptions here
        }
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
```

16. Change the value for `int i` and `int j` to other numbers, such as 3 and 4.
17. Add a line that prints out an exception, if an exception is thrown. The `try`/`catch` block is follows (new and changed lines from this step and the previous step are highlighted in bold text):

```
try { // Call Web Service Operation
    org.me.calculator.client.CalculatorWS port =
        service.getCalculatorWSPort();
    // TODO initialize WS operation arguments here
    int i = 3;
    int j = 4;
    // TODO process result here
    int result = port.add(i, j);
    out.println("<p>Result: " + result);
} catch (Exception ex) {
    out.println("<p>Exception: " + ex);
}
```

18. If Reliable Messaging is enabled, the client needs to close the port when done or the server log will be overwhelmed with messages. To close the port, first add the following line to the import statements at the top of the file:

```
import com.sun.xml.ws.Closeable;
```

Then add the line in bold at the end of the `try` block, as shown below.

```
try { // Call Web Service Operation
    org.me.calculator.client.CalculatorWS port =
        service.getCalculatorWSPort();
    // TODO initialize WS operation arguments here
    int i = 3;
    int j = 4;
    // TODO initialize WS operation arguments here
    int i = 3;
    int j = 4;
    // TODO process result here
    int result = port.add(i, j);
```

```
        out.println("<p>Result: " + result);
        ((Closeable)port).close();
    } catch (Exception ex) {
        out.println("<p>Exception: " + ex);
    }
```

19. Right-click the project node and choose Run Project. The server starts (if it was not running already), the application is built, deployed, and run. The browser opens and displays the calculation result.

You have successfully created and deployed a WSIT-enabled client that can access a WSIT-enabled web service.

3 Bootstrapping and Configuration

THIS chapter explains how to retrieve information that is used to access and consume a WSIT-enabled web service and provides pointers to examples that demonstrate how to bootstrap and configure WSIT-enabled clients from Web Services Description Language (WSDL) files.

The following topics are covered in this chapter:

- What is a Server-Side Endpoint? (page 33)
- Creating a Client from WSDL (page 34)
- Client From WSDL Examples (page 35)

What is a Server-Side Endpoint?

Web services expose one or more endpoints to which messages can be sent. A web service endpoint is an entity, processor, or resource that can be referenced and to which web services messages can be addressed. Endpoint references convey the information needed to address a web service endpoint. Clients need to know this information before they can access a service.

Typically, web services package endpoint descriptions and use a WSDL file to share these descriptions with clients. Clients use the web service endpoint description to generate code that can send SOAP messages to and receive SOAP messages from the web service endpoint.

Creating a Client from WSDL

To create a web service client that can access and consume a web service provider, you must obtain the information that defines the interoperability requirements of the web service provider. Providers make this information available by means of WSDL files. WSDL files may be made available in service registries or published on the Internet via a URL (or both). You can use a web browser or the Netbeans IDE to obtain WSDL files.

A WSDL file contains descriptions of the following:

- **Network services**: The description includes the name of the service, the location of the service, and ways to communicate with the service, that is, what transport to use.
- **Web services policies**: Policies express the capabilities, requirements, and general characteristics of a web service. Web service providers use policies to specify policy information in a standardized way. Policies convey conditions on interactions between two web service endpoints. Typically, the provider of a web service exposes a policy to convey conditions under which it provides the service. A requester (a client) might use the policy to decide whether or not to use the service.

Web Services Metadata Exchange (WS-MEX) is the protocol for requesting and transferring the WSDL from the provider to the client. This protocol is a bootstrap mechanism for communication. When the type of metadata desired is clearly known (for example, WS-Policy), a client request may indicate that only that type should be returned.

Client From WSDL Examples

The following sections, found in other chapters of this tutorial, explain how to create a client from a WSDL file using the example files in the tutorial bundle:

- Creating a Client to Consume a WSIT-Enabled Web Service (page 29) shows how to create a client from WSDL using a web container and the NetBeans IDE.
- Creating a Client from WSDL (page 138) shows how to create a client from WSDL using only a web container.

4

Message Optimization

THIS chapter provides instructions on how to configure message optimization in web service providers and clients.

Note: Because of the special encoding/decoding requirements for message optimization, if a service uses message optimization, then a client of that service must support message optimization. Most web services stacks do support message optimization. In the rare case when you think that a legacy client, which does not support optimization, will access your service, do not use message optimization. In general, however, it is a safe and good practice to use message optimization.

This chapter covers the following topics:

Creating a Web Service

The starting point for developing a web service to use the WSIT technologies is a Java class file annotated with the `javax.jws.WebService` annotation.

For detailed instructions for how to use NetBeans IDE to create a web service, see Creating a Web Service (page 24).

Configuring Message Optimization in a Web Service

To use the IDE to configure a web service for message optimization, perform the following steps:

1. In the IDE Projects window, expand the Web Services node, right-click the CalculatorWS node, and choose Edit Web Service Attributes, as shown in Figure 4–1. The Web Service Attributes editor appears.

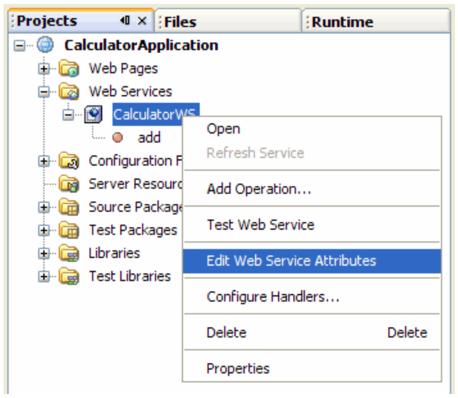
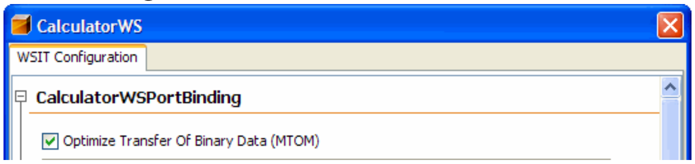


Figure 4–1 Selecting the Edit Web Services Attributes Option

2. Select the Optimize Transfer of Binary Data (MTOM) check box, as shown in Figure 4–2, and click Ok.



[✓] Optimize Transfer Of Binary Data (MTOM)

Figure 4–2 Enabling MTOM

This setting configures the web service to optimize messages that it transmits and to decode optimized messages that it receives.

Deploying and Testing a Web Service

Now that you have configured the web service to use message optimization, you can deploy and test it.

To deploy and test the web service, perform the following steps:

1. Right-click the project node, select Properties, and select Run.
2. Type `/CalculatorWSService?wsdl` in the Relative URL field and click OK.
3. Right-click the project node and choose Run Project. The IDE starts the web container, builds the application, and displays the WSDL file page in your browser.

You have now successfully tested the deployment of a web service with message optimization enabled.

Creating a Client to Consume a WSIT-enabled Web Service

Now that you have built and tested a web service that uses the WSIT Message Optimization technology, you can create a client that accesses and consumes that web service. The client will use the web service's WSDL to create the functionality necessary to satisfy the interoperability requirements of the web service.

To create a client to access and consume the web service, perform the following steps:

1. Choose File→New Project, select Web Application from the Web category and click Next.
2. Name the project, for example, CalculatorWSServletClient.
3. Make sure that the J2EE version is set to Java EE 5, then click Finish.
4. Right-click the CalculatorWSServletClient node and select New→Web Service Client. The New Web Service Client window appears.
5. Cut and paste the URL of the web service that you want the client to consume into the WSDL URL field, for example, `http://localhost:8080/`

`CalculatorApplication/CalculatorWSService?wsdl`, the URL of the CalculatorWS web service.

6. Type `org.me.calculator.client` in the Package field, and click Finish. The Projects tab displays the new web service client, shown in Figure 4–3.

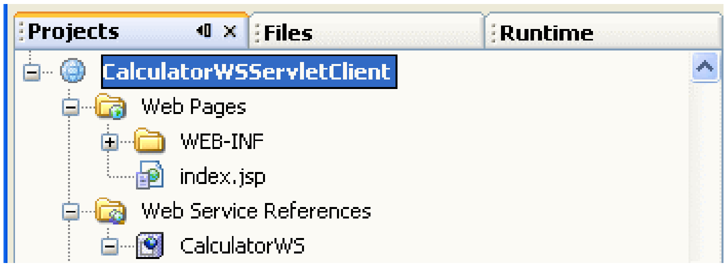


Figure 4–3 Web Service Client

7. Right-click the CalculatorWSServletClient project node and choose New→Servlet.
8. Name the servlet ClientServlet, specify the package name, for example, `org.me.calculator.client` and click Finish.
9. To make the servlet the entry point to your application, right-click the project node, choose Properties, click Run, type `/ClientServlet` in the Relative URL field and click OK.
10. Double-click `ClientServlet.java` so that it opens in the Source Editor.
11. In the Source Editor, remove the line that comments out the body of the processRequest method. This is the start-comment line that starts the section that comments out the code:

    ```
    /* TODO output your page here
    ```
12. Delete the end-comment line that ends the section of commented out code:

    ```
    */
    ```
13. Add some empty lines after the following line:

    ```
    out.println("<h1>Servlet ClientServlet at " +
          request.getContextPath () + "</h1>");
    ```
14. Right-click in one of the empty lines that you added. Choose Web Service Client Resources→Call Web Service Operation. The Select Operation to Invoke dialog box appears.
15. Browse to the Add operation and click OK. The `processRequest` method looks as follows (the added code is in bold below):

```
protected void processRequest(HttpServletRequest
    request, HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html;charset=UTF-8");
  PrintWriter out = response.getWriter();
  out.println("<html>");
  out.println("<head>");
  out.println("<title>Servlet ClientServlet</title>");
  out.println("</head>");
  out.println("<body>");
  out.println("<h1>Servlet ClientServlet at "
      + request.getContextPath () + "</h1>");
  try { // Call Web Service Operation
    org.me.calculator.client.CalculatorWS port =
      service.getCalculatorWSPort();
    // TODO initialize WS operation arguments here
    int i = 0;
    int j = 0;
    // TODO process result here
    int result = port.add(i,  j);
    system.out.println("Result = "+result);
  } catch (Exception ex) {
    // TODO handle custom exceptions here
  }
  out.println("</body>");
  out.println("</html>");
  out.close();
}
```

16. Change the value for `int i` and `int j` to other numbers, such as 3 and 4.
17. Change the `System.out.println` statement to `out.println`.
18. Add a line that prints out an exception, if an exception is thrown. The `try/catch` block should look as follows (new and changed lines are highlighted in bold text):

```
try { // Call Web Service Operation
  org.me.calculator.client.CalculatorWS port =
    service.getCalculatorWSPort();
// TODO initialize WS operation arguments here
int i = 3;
int j = 4;
// TODO process result here
int result = port.add(i, j);
```

```
        out.println("<p>Result: " + result);
    } catch (Exception ex) {
        out.println("<p>Exception: " + ex);
    }
```

19. Right-click the project node and choose Run Project. The server starts (if it was not running already) the application is built and deployed, and the browser opens and displays the calculation result.

You have successfully created and deployed a client that can access a web service with message optimization enabled.

Message Optimization and Secure Conversation

The Web Services Secure Conversation technology has message optimization benefits. While providing better message-level security it also improves the efficiency of multiple-message exchanges. It accomplishes this by providing basic mechanisms on top of which secure messaging semantics can be defined for multiple-message exchanges. This feature allows for contexts to be established so that potentially more efficient keys or new key material can be exchanged. The result is that the overall performance of subsequent message exchanges is improved.

For more information on how to use Secure Conversation, see Chapter 6.

5

Using Reliable Messaging

THIS chapter explains how to configure reliable messaging in web service providers and clients.

This chapter covers the following topics:

- Reliable Messaging Options (page 43)
- Creating Web Service Providers and Clients that use Reliable Messaging (page 45)
- Using Secure Conversation With Reliable Messaging (page 45)

Reliable Messaging Options

Table 5–1 describes the reliable messaging configuration options.

Table 5–1 Endpoint Reliable Messaging Configuration Options

Option	**Description**
Reliable Messaging	Specifies whether reliable messaging is enabled.

Table 5–1 Endpoint Reliable Messaging Configuration Options

Option	**Description**
Ordered Delivery	Specifies whether the Reliable Messaging protocol ensures that the application messages for a given message sequence are delivered to the endpoint application in the order indicated by the message numbers. This option increases the time to process application message sequences and may result in the degradation of web service performance. Therefore, you should not enable this option unless ordered delivery is required by the web service.
Flow Control	Specifies whether the Flow Control feature is enabled. When enabled, this option works in conjunction with the Max Buffer Size setting to determine the maximum number of messages for sequence that can be stored at the endpoint awaiting delivery to the application. Messages may have to be withheld from the application if ordered delivery is required and some of their predecessors have not arrived. If the number of stored messages reaches the threshold specified in the Max Buffer Size setting, incoming messages belonging to the sequence are ignored.
Max Buffer Size	If Flow control is enabled, specifies the number of messages that will be buffered for a message sequence. The default setting is 32. For more information, see the description of the Flow Control option.
Inactivity Timeout	Specifies the time interval beyond which either source or destination may terminate any message sequence due to inactivity. The default setting is 600,000 milliseconds (10 minutes). A web service endpoint will always terminate a sequence whose inactivity timeout has expired. To keep the sequence active, an inactive client will always send a stand- alone message with an `AckRequested` header to act as a heartbeat as the end of the Inactivity timeout interval approaches.

Creating Web Service Providers and Clients that use Reliable Messaging

Examples and detailed instructions on how to create web service providers and clients that use reliable messaging are provided in Chapters 2 and 7 of this tutorial.

- For an example of creating a web service and a client using a web container and NetBeans IDE, see Chapter 2.
- For an example of creating a web service and a client using only a web container, see Chapter 7.

Using Secure Conversation With Reliable Messaging

If Secure Conversation is enabled for the web service endpoint, the web service acquires a Security Context Token (SCT) for each application message sequence, that is, each message sequence is assigned a different SCT. The web service then uses that token to sign all messages exchanged for that message sequence between the source and destination for the life of the sequence. Hence, there are two benefits in using Secure Conversation with Reliable Messaging:

- The sequence messages are secure while in transit between the source and destination endpoints.
- If there are different users accessing data at the source and destination endpoints, the SCT prevents users from seeing someone else's data.

Note: Secure Conversation is a WS-Security option, not a reliable messaging option. If Secure Conversation is enabled on the web service endpoint, Reliable Messaging uses Security Context Tokens.

For more information on how to use Secure Conversation, see Chapter 6.

6

Using WSIT Security

THIS chapter describes how to use NetBeans Integrated Development Environment (the IDE) to configure security for web services and web service clients using WSIT.

This release of WSIT makes securing web services even easier by including a set of preconfigured security mechanisms that can be applied to a web service or a web service operation simply by selecting it from a list. You can use advanced configuration options to customize the security mechanism to the needs of your application.

This chapter covers the following topics:

Configuring Security Using NetBeans IDE

This section contains the following topics:

- Securing the Service (page 48)
- Securing the Client (page 50)

Securing the Service

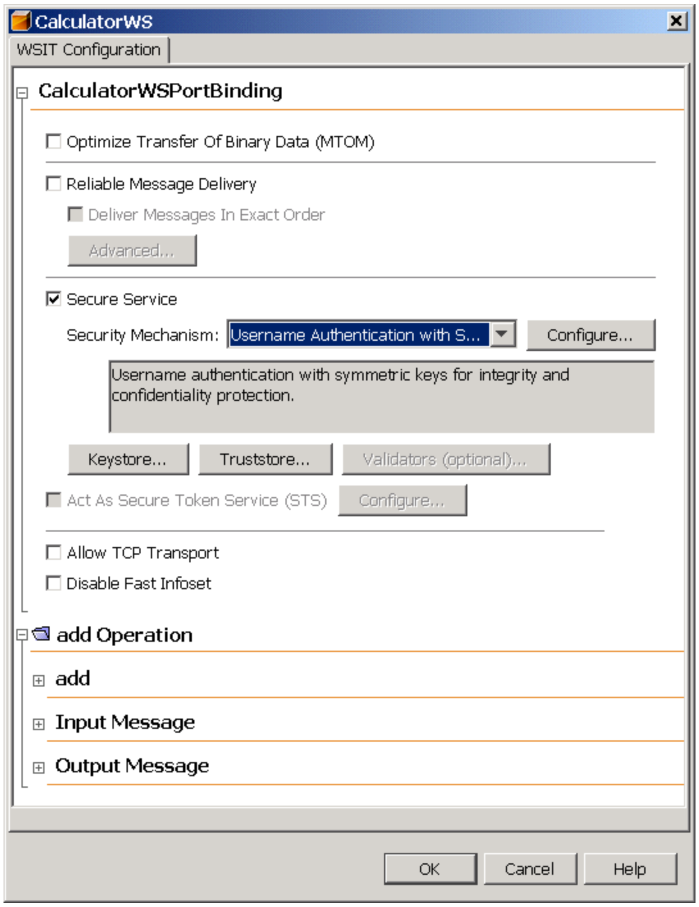
To use the IDE to configure security for a web service and/or a web service operation, perform the following tasks:

1. Create or open your web service.

   If you need an example of how to create a web service, refer to Chapter 2, WSIT Example Using a Web Container and NetBeans.

   NOTE: When creating an application using the wizards in NetBeans and running on GlassFish, the Java EE Version defaults to Java EE 5. This results in an application compliant with JSR-109, *Implementing Enterprise Web Services*, which can be read at `http://jcp.org/en/jsr/detail?id=109`. If you select a value other than the default, for example, J2EE 1.4, the application that is created is not JSR-109 compliant, which means that the application is not JAX-WS, but is JAX-RPC.
2. In the Projects window, expand the Web Services node.
3. Right-click the node for the web service you want to secure.
4. Select Edit Web Service Attributes.

   When the Web Service Attributes Editor is opened, the WSIT Configuration options display (see Figure 6–1).



Username authentication with symmetric keys for integrity and confidentiality protection.

Figure 6–1 Web Service Attributes Editor Page

5. Select **Secure Service**. This option enables WSIT security for all of the operations of a web service.

   For information on how to secure selected operations, refer to Securing an Operation (page 85).

6. Select a **Security Mechanism** from the list.

   Most of the mechanisms are fully functional without further configuration, however, if you'd like to customize the mechanism, click **Configure** to specify the configuration for that mechanism.

7. Specify **Keystore**, **Truststore**, STS, SSL, and/or user information as required for the selected security mechanism.

   Refer to the entry for the selected security mechanism in Table 6–1 for further information. This table summarizes the information that need to be set up for each of the security mechanisms.

8. Click OK to save your changes.
9. Run the web application by right-clicking the project node and selecting Run Project.
10. Verify the URL of the WSDL file before proceeding with the creation of the web service client:

    The client will be created from this WSDL file, and will get the service's security policies through the web service reference URL when the client is built or refreshed.

The WSIT Configuration file that is used when the web service is deployed can be viewed by expanding the Web Pages→WEB-INF elements of the application in the tree, and then double-clicking the `wsit-<`*`package`*`>.<`*`service`*`>.xml` file to open it in the editor.

Steps for configuring an example application are provided for several of the mechanisms. Please see the following sections for a complete example of how to configure a web service and a web service client to use these security mechanisms:

- Example: Username Authentication with Symmetric Keys (UA) (page 97)
- Example: Mutual Certificates Security (MCS) (page 100)
- Example: Transport Security (SSL) (page 103)
- Example: SAML Authorization over SSL (SA) (page 106)
- Example: SAML Sender Vouches with Certificates (SV) (page 111)
- Example: STS Issued Token (STS) (page 115)

Securing the Client

All of the steps in Securing the Service (page 48) need to be completed before you create your web service client. The service's security policies are defined in its WSDL. You specify this WSDL file when you create the client application so that the client is configured to work with the service's security mechanism through the web service reference URL when the client is built or refreshed.

To use the IDE to configure security for a web service client, perform the following tasks:

1. Create a client for your web service.

   If you need an example of how to do this, see Creating a Client to Consume a WSIT-Enabled Web Service (page 29).

   If you are creating a client for a mechanism that will use **SSL**, specify the secure port for running the client when completing the New Web Service Client step. To do this, enter `https://<`*`fully_qualified_hostname`*`>:8181/<`*`rest_of_url`*`>` in the WSDL URL field of the New Web Service Client wizard. For the example, this is the way to specify the secure URL for CalculatorWS web service:
   `https://<`*`fully_qualified_hostname`*`>:8181/CalculatorApplication/CalculatorWSService?wsdl`

   NOTE: If you prefer to use `localhost` in place of the fully-qualified hostname when specifying the URL for the service WSDL, you must follow the workaround described in Transport Security (SSL) Workaround (page 63).

2. In the Projects window, expand the client node.
3. Expand the Web Service References node.
4. Right-click the node for the web service reference you want to secure.
5. Select Edit Web Service Attributes.

   When the Web Service References Attributes Editor is opened, select the WSIT tab to display the WSIT options (see Figure 6–2).

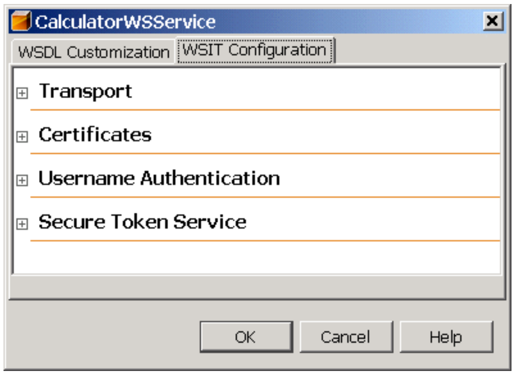


Figure 6–2 Web Service References Attributes Editor Page for Web Service Clients

6. Refer to Table 6–2 for a summary of what options are required on the client side. The configuration requirements for the client are dependent upon which security mechanism is specified on the server side.
7. Click OK to save your changes.

   This information is saved in a WSDL file under Source Packages→META-INF.

To view the WSDL file containing the WSIT security elements, in the tree, drill down from the project to Source Packages→META-INF, and then double-click on <*service*>Service.wsdl.

Summary of Configuration Requirements

The following sections summarize the options that need to be configured for each of the security mechanisms on both the service and client side. The configuration requirements for the client are dependent upon which security mechanism is specified on the server side.

The following sections assume that you have completed the steps described in Configuring SSL and Authorized Users (page 68) for creating an authorized user and setting up your system for SSL and Updating GlassFish Certificates (page 74) for upgrading the GlassFish certificates if the security mechanism you are using requires these things.

Summary of Service-Side Configuration Requirements

Table 6–1 summarizes the options that need to be configured for each of the security mechanisms. Each of the columns is briefly discussed after the table.

Table 6–1 Summary of Service-Side Configuration Requirements

Mechanism	Keystore	Truststore	STS	SSL	User in GlassFish
Username Auth. w/Symmetric Keys	YES				YES
Mutual Certs.	YES	YES (no alias)			
Transport Sec.				YES	YES
Message Auth. over SSL - User-name Token				YES	YES
Message Auth. over SSL - X.509 Token		YES (no alias)		YES	
SAML Auth. over SSL	YES	YES (no alias)		YES	YES
Endorsing Cert.	YES				
SAML Sender Vouches with Cert.	YES	YES (no alias)			YES
SAML Holder of Key	YES	YES (no alias)			YES
STS Issued Token			YES		
STS Issued Token with Ser-vice Cert.			YES		
STS Issued Endorsing Token			YES		

- **Keystore**—If this column indicates YES, click the Keystore button and configure the keystore to specify the alias identifying the service certificate and private key. For the GlassFish keystores, the file is `keystore.jks` and the alias is `xws-security-server`, assuming that you've updated the GlassFish default certificate stores as described in Updating GlassFish Certificates (page 74).
- **Truststore**—If this column indicates YES, click the Truststore button and configure the truststore to specify the alias that contains the certificate and trusted roots of the client. For the GlassFish keystores, the file is `cacerts.jks` and the alias is `xws-security-client`, assuming that you've updated the GlassFish default certificate stores as described in Updating GlassFish Certificates (page 74).
- **STS**—If this column indicates YES, you must have a Security Token Service that can be referenced by the service. An example of an STS can be found in the section Creating and Securing the STS (STS) (page 117). The STS is secured using a separate (non-STS) security mechanism. The security configuration for the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application.
- **SSL**—To use a mechanism that uses secure transport (SSL), you must configure the *system* to point to the client and server keystore and truststore files. Steps for doing this are described in Configuring SSL For Your Applications (page 69).
- **User in Glassfish**—To use a mechanism that requires a user database for authentication, you can add a user to the file realm of GlassFish. Instructions for doing this can be found at Adding Users to GlassFish (page 72).

Summary of Client-Side Configuration Requirements

Table 6–2 summarizes the options that need to be configured for each of the security mechanisms on the client-side. Each of the columns is briefly discussed after the table.

Table 6–2 Summary of Client-Side Configuration Requirements

Mechanism	Key store	Trust store	Default User	SAML Callback Handler	STS	SSL	User in GF
Username Auth. w/Symmetric Keys		YES	YES				YES
Mutual Certs.	YES	YES					
Transport Sec.						YES	YES
Message Auth. over SSL - Username Token			YES			YES	YES
Message Auth. over SSL - X.509 Token	YES					YES	
SAML Auth. over SSL	YES	YES		YES		YES	YES
Endorsing Cert.	YES	YES					
SAML Sender Vouches with Cert.	YES	YES		YES			YES
SAML Holder of Key	YES	YES		YES			YES
STS Issued Token	YES	YES			Y		
STS Issued Token with Service Cert.	YES	YES			Y		

Table 6–2 Summary of Client-Side Configuration Requirements (Continued)

Mechanism	Key store	Trust store	Default User	SAML Callback Handler	STS	SSL	User in GF
STS Issued Endorsing Token	YES	YES			Y		

- **Keystore**—If this column indicates YES, configure the keystore to point to the alias for the client certificate. For the GlassFish keystores, the keystore file is `keystore.jks` and the alias is `xws-security-client`, assuming that you've updated the GlassFish default certificate stores as described in Updating GlassFish Certificates (page 74).
- **Truststore**—If this column indicates YES, configure the truststore that contains the certificate and trusted roots of the server. For the GlassFish keystores, the file is `cacerts.jks` and the alias is `xws-security-server`, assuming that you've updated the GlassFish default certificate stores as described in Updating GlassFish Certificates (page 74).

  When using an STS mechanism, the client specifies the truststore and certificate alias for the STS, not the service. For the GlassFish stores, the file is `cacerts.jks` and the alias is `wssip`.
- **Default User**—When this column indicates YES, you must configure either a default username and password, a UsernameCallbackHandler, or leave these options blank and specify a user at runtime. More information on these options can be found at Configuring Username Authentication on the Client (page 57).
- **SAML Callback Handler**—When this column indicates YES, you must specify a SAML Callback Handler. Examples of SAML Callback Handlers are described in Example SAML Callback Handlers (page 59).
- **STS**—If this column indicates YES, you must have a Security Token Service that can be referenced by the service. An example of an STS can be found in the section Creating and Securing the STS (STS) (page 117). The STS is secured using a separate (non-STS) security mechanism. The security configuration for the client-side of this application is dependent upon

the security mechanism selected for the STS, and not on the security mechanism selected for the application.

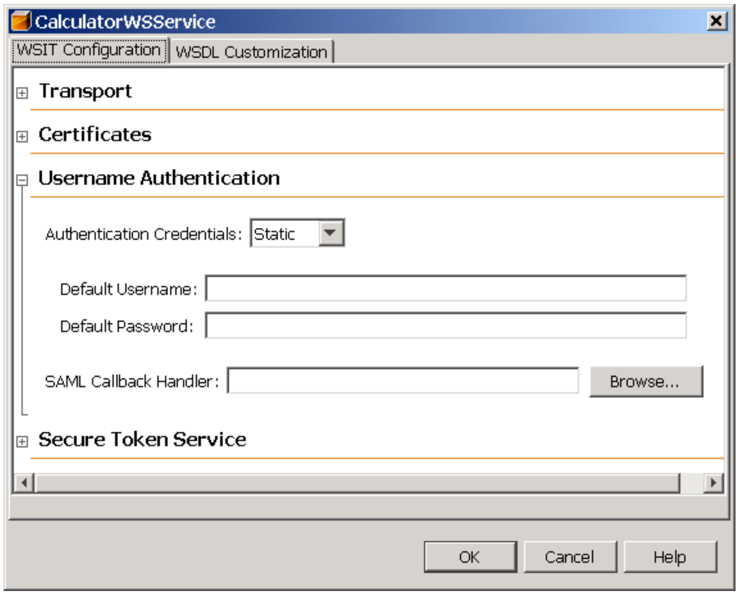
- **SSL**—To use a mechanism that uses secure transport (SSL), you must configure the system to point to the client and server keystore and truststore files. Steps for doing this are described in Configuring SSL For Your Applications (page 69).
- **User in Glassfish**—To use a mechanism that requires a user database for authentication, you can add a user to the `file` realm of GlassFish. Instructions for doing this can be found at Adding Users to GlassFish (page 72).

Configuring Username Authentication on the Client

On the client side, a user name and password must be configured for some of the security mechanisms. For this purpose, you can use the default Username and Password Callback Handlers (when deploying to GlassFish), specify a SAML Callback Handler, specify a default user name and password for development purposes, create and specify your own Callback Handlers if the container you are using does not provide defaults, or leave all of these options blank and specify the username and password dynamically at runtime. When using any of these options, you must create an authorized user on GlassFish using the Admin Console, as described in Adding Users to GlassFish (page 72).

Once you've created an authorized user and determined how your application needs to specify the user, configure the Username Authentication options, as follows:

1. In the Projects window, expand the node for the web service client.
2. Expand the Web Service References node.
3. Right-click the node for the web service reference for which you want to configure security options.
4. Select Edit Web Service Attributes.
5. Select the WSIT tab to display the WSIT options.
6. Expand the Username Authentication section to specify the user name and password information as required by the service. The dialog displays as follows:



Authentication Credentials: Static

Default Username:

Default Password:

SAML Callback Handler:

Figure 6–3 WSIT Configuration - Client - Username Authentication

7. The following options are available:

   NOTE: Currently the GlassFish `CallbackHandler` cannot handle the following: SAML Callbacks and Require ThumbPrint Reference assertions under an X.509 Token. This may be addressed in a future milestone.

- **Authentication Credentials**—Select Static or Dynamic.
- **Default Username**, **Default Password**—Enter the name of an authorized user and the password for this user. This option is best used only in the development environment. When the Default Username and Default Password are specified, the username and password are stored in the `wsit-client.xml` file in clear text, which presents a security risk. Do not use this option for production.
- **SAML Callback Handler**—To use a SAML Callback Handler, you need to create one, as there is no default. References to example SAML Callback Handlers are provided in Example SAML Callback Handlers (page 59). An example that uses a SAML Callback Handler can be found in Example: SAML Authorization over SSL (SA) (page 106).

Example SAML Callback Handlers

Creating a SAML Callback Handler is beyond the scope of this document, however, the following web pages may be helpful for this purpose:

- A client-side configuration, which includes a SAML Callback Handler, can be viewed at the following URL:
  `https://wsit.dev.java.net/source/browse/*checkout*/wsit/wsit/test/e2e/testcases/xwss/s11/resources/wsit-client.xml`
- An example of a SAML Callback Handler can be viewed and/or downloaded from the following URL:
  `https://xwss.dev.java.net/servlets/ProjectDocumentList?folderID=6645&expandFolder=6645&folderID=6645`
- An example application in this tutorial that uses a SAML Callback Handler can be found in Example: SAML Authorization over SSL (SA) (page 106).

When writing SAML Callback Handlers for different security mechanisms, set the subject confirmation method to SV (Sender Vouches) or HOK (Holder of Key) and the appropriate SAML Assertion version depending on the SAML version and SAML Token Profile selected when setting the security mechanism for the service.

For example, the following code snippet for one of the SAMLCallbackHandlers listed above demonstrates how to set the subject confirmation method and sets the SAMLAssertion version to 1.0, profile 1.0.

```
if(callbacks[i] instanceof SAMLCallback) {
  try {

     SAMLCallback samlCallback = (SAMLCallback)callbacks[i];

     /*
     Set confirmation Method to SV [SenderVouches] or
     HOK[Holder of Key]
     */
     samlCallback.setConfirmationMethod
        (samlCallback.SV_ASSERTION_TYPE);

     if(samlCallback.getConfirmationMethod().equals(
           samlCallback.SV_ASSERTION_TYPE)) {
        samlCallback.setAssertionElement
           (createSVSAMLAssertion());

        svAssertion_saml10 =
           samlCallback.getAssertionElement();
```

```
            /*
            samlCallback.setAssertionElement
              (createSVSAMLAssertion20());
            svAssertion_saml20 =
              samlCallback.getAssertionElement();
            */
        }else

          if(samlCallback.getConfirmationMethod().equals(
              samlCallback.HOK_ASSERTION_TYPE)) {
            samlCallback.setAssertionElement
              (createHOKSAMLAssertion());
            hokAssertion_saml10 =
              samlCallback.getAssertionElement();
            /*
            samlCallback.setAssertionElement
              (createHOKSAMLAssertion20());
            hokAssertion_saml20 =
              samlCallback.getAssertionElement();
            */
        }

      } catch (Exception e) {
        e.printStackTrace();
        }
      } else {
        throw unsupportedCallback;
      }
```

Security Mechanisms

The following lists the possible choices for security mechanisms. These mechanisms are discussed in the following sections:

- Username Authentication with Symmetric Keys (page 61)
- Mutual Certificates Security (page 62)
- Transport Security (SSL) (page 62)
- Message Authentication over SSL (page 64)
- SAML Authorization over SSL (page 64)
- Endorsing Certificate (page 65)
- SAML Sender Vouches with Certificates (page 65)
- SAML Holder of Key (page 66)
- STS Issued Token (page 66)
- STS Issued Token with Service Certificate (page 67)
- STS Issued Endorsing Token (page 67)

A table that summarizes the configuration options on the server side is available in Summary of Service-Side Configuration Requirements (page 52).

Username Authentication with Symmetric Keys

The Username Authentication with Symmetric Keys mechanism protects your application for integrity and confidentiality. Symmetric key cryptography relies on a single, shared secret key that is used to both sign and encrypt a message. Symmetric keys are usually faster than public key cryptography.

For this mechanism, the client does not possess any certificate/key of his own, but instead sends its username/password for authentication. The client shares a secret key with the server. The shared, symmetric key is generated at runtime and encrypted using the service's certificate. The client must specify the alias in the truststore by identifying the server's certificate alias.

See Also: Example: Username Authentication with Symmetric Keys (UA) (page 97).

Mutual Certificates Security

The Mutual Certificates Security mechanism adds security via authentication and message protection that ensures integrity and confidentiality. When using mutual certificates, a keystore and truststore file must be configured for both the client and server sides of the application.

See Also: Example: Mutual Certificates Security (MCS) (page 100).

Transport Security (SSL)

The Transport Security mechanism protects your application during transport using SSL for authentication and confidentiality. Transport-layer security is provided by the transport mechanisms used to transmit information over the wire between clients and providers, thus transport-layer security relies on secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL). Transport security is a point-to-point security mechanism that can be used for authentication, message integrity, and confidentiality. When running over an SSL-protected session, the server and client can authenticate one another and negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. Security is “live” from the time it leaves the consumer until it arrives at the provider, or vice versa. The problem is that it is not protected once it gets to its destination. For protection of data after it reaches its destination, use one of the security mechanisms that uses SSL and also secures data at the message level.

Digital certificates are necessary when running secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL). The HTTPS service of most web servers will not run unless a digital certificate has been installed. Digital certificates have already been created for GlassFish, and the default certificates are sufficient for running this mechanism, and are required when using Atomic Transactions (see Chapter 10). However, the message security mechanisms require a newer version of certificates than is available with GlassFish. You can download valid keystore and truststore files for the client and server as described in Updating GlassFish Certificates (page 74).

To use this mechanism, follow the steps in Configuring SSL For Your Applications (page 69).

See Also: Example: Transport Security (SSL) (page 103).

Transport Security (SSL) Workaround

This note applies to cases where `https` is the transport protocol used between a WSIT client and a secure web service using transport binding, and you are referencing `localhost` when creating the client.

Note: If you use the fully-qualified hostname (FQHN) in the URL for the service WSDL when you are adding the web service client to the client application, this workaround is not required. It is only required when you specify `localhost` in the URL for the service WSDL.

During *development* (not production) it is sometimes convenient to use certificates whose CN (Common Name) does *not* match the host name in the URL.

A developer would want to use a CN which is different from the host name in the URL in WSIT when using `https` addresses in Dispatch clients and during `wsimport`. The below mentioned workaround is only for the Dispatch clients, which are also used in WS-Trust to communicate with STS. This has to be done even if the client's main service is not on `https`, but only the STS is on `https`.

Java by default verifies that the certificate CN (Common Name) is the same as host name in the URL. If the CN in the certificate is not the same as the host name, your web service client fails with the following exception:

```
javax.xml.ws.WebServiceException: java.io.IOException: HTTPS
hostname wrong: should be <hostname as in the certificate>
```

The recommended way to overcome this issue is to generate the server certificate with the Common Name (CN) matching the host name.

To workaround this only during development, in your client code, you can set the default host name verifier to a custom host name verifier which does a custom check. An example is given below. It is sometimes necessary to include this in the static block of your main Java class as shown below to set this verifier before any connections are made to the server.

```
static {
    //WORKAROUND. TO BE REMOVED.


javax.net.ssl.HttpsURLConnection.setDefaultHostnameVerifier(
     new javax.net.ssl.HostnameVerifier(){
  public boolean verify(String |hostname|,
    javax.net.ssl.SSLSession sslSession) {
    if (hostname.equals("mytargethostname")) {
      return true;
    }
     return false;
  }
});
}
```

Please remember to remove this once you install valid certificates on the server.

Message Authentication over SSL

The Message Authentication over SSL mechanism attaches a cryptographically secured identity or authentication token with the message and use SSL for confidentiality protection.

By default, a Username Supporting Token will be used for message authentication. To use an X.509 Supporting Token instead, click the Configure button and select X509. Under this scenario, you will need to configure your system for using SSL as described in Configuring SSL For Your Applications (page 69).

SAML Authorization over SSL

The SAML Authorization over SSL mechanism attaches an authorization token with the message and uses SSL for confidentiality protection. In this mechanism, the SAML token is expected to carry some authorization information about an end user. The sender of the token is actually vouching for the credentials in the SAML token.

To use this mechanism, configure SSL on the server, as described in Configuring SSL For Your Applications (page 69), and, on the clients side, configure a **`SAMLCallbackHandler`** as described in Example SAML Callback Handlers (page 59).

See Also: Example: SAML Authorization over SSL (SA) (page 106).

Endorsing Certificate

This mechanism uses secure messages using symmetric key for integrity and confidentiality protection, and uses an endorsing client certificate to augment the claims provided by the token associated with the message signature. For this mechanism, the client knows the service's certificate, and requests need to be endorsed/authorized by a special identity. For example, all requests to a vendor must be endorsed by a purchase manager, so the certificate of the purchase manager should be used to endorse (or counter sign) the original request.

SAML Sender Vouches with Certificates

This mechanism protects messages with mutual certificates for integrity and confidentiality and with a Sender Vouches SAML token for authorization. The Sender Vouches method establishes the correspondence between a SOAP message and the SAML assertions added to the SOAP message. The attesting entity provides the confirmation evidence that will be used to establish the correspondence between the subject of the SAML subject statements (in SAML assertions) and SOAP message content. The attesting entity, presumed to be different from the subject, vouches for the verification of the subject. The receiver has an existing trust relationship with the attesting entity. The attesting entity protects the assertions (containing the subject statements) in combination with the message content against modification by another party. For more information about the Sender Vouches method, read the SAML Token Profile document at `http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf`.

For this mechanism, the SAML token is included as part of the message signature as an authorization token and is sent only to the recipient. The message payload needs to be signed and encrypted. The requestor is vouching for the credentials (present in the SAML assertion) of the entity on behalf of which the requestor is acting.

The initiator token, which is an X.509 token, is used for signature. The recipient token, which is also an X.509 token, is used for encryption. For the server, this is reversed, the recipient token is the signature token and the initiator token is the encryption token. A SAML token is used for authorization.

See Also: Example: SAML Sender Vouches with Certificates (SV) (page 111).

SAML Holder of Key

This mechanism protects messages with a signed SAML assertion (issued by a trusted authority) carrying client public key and authorization information with integrity and confidentiality protection using mutual certificates. The Holder-of-Key (HOK) method establishes the correspondence between a SOAP message and the SAML assertions added to the SOAP message. The attesting entity includes a signature that can be verified with the key information in the confirmation method of the subject statements of the SAML assertion referenced for key info for the signature. For more information about the Holder of Key method, read the SAML Token Profile document at `http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf`.

Under this scenario, the service does not trust the client directly, but requires the client to send a SAML assertion issued by a particular SAML authority. The client knows the recipient's public key, but does not share a direct trust relationship with the recipient. The recipient has a trust relationship with the authority that issues the SAML token. The request is signed with the client's private key and encrypted with the server certificate. The response is signed using the server's private key and encrypted using the key provided within the HOK SAML assertion.

STS Issued Token

This security mechanism protects messages using a token issued by a trusted Secure Token Service (STS) for message integrity and confidentiality protection.

An STS is a service that implements the protocol defined in the WS-Trust specification (you can find a link to this specification at `https://wsit.dev.java.net/`.) This protocol defines message formats and message exchange patterns for issuing, renewing, canceling, and validating security tokens.

Service providers and consumers are in potentially different managed environments but use a single STS to establish a chain of trust. The service does not trust the client directly, but instead trusts tokens issued by a designated STS. In other words, the STS is taking on the role of a second service with which the client has to securely authenticate. The issued tokens contain a key, which is encrypted for the server and which is used for deriving new keys for signing and encrypting.

To use this mechanism for the web service, you simply select this option as your security mechanism. However, you must have a Security Token Service that can be referenced by the service. An example of an STS can be found in the section Creating and Securing the STS (STS) (page 117). In this section, you select a security mechanism for the STS. The security configuration for the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application. The client trust-store must contain the certificate of the STS, which has the alias of `wssip` if you are using the updated GlassFish certificates.

See Also: Example: STS Issued Token (STS) (page 115).

STS Issued Token with Service Certificate

This security mechanism is similar to the one discussed in STS Issued Token (page 66), with the difference being that in addition to the service requiring the client to authenticate using a SAML token issued by a designated STS, confidentiality protection is achieved using a service certificate. A service certificate is used by a client to authenticate the service and provide message protection. For GlassFish, a default certificate of `s1as` is installed.

To use this mechanism for the web service, you simply select this option as your security mechanism. However, you must have a Security Token Service that can be referenced by the service. An example of an STS can be found in the section Creating and Securing the STS (STS) (page 117). In this section, you select a security mechanism for the STS. The security configuration for the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application. The client trust-store must contain the certificate of the STS, which has the alias of `wssip` if you are using the updated GlassFish certificates.

STS Issued Endorsing Token

This security mechanism is similar to the one discussed in STS Issued Token (page 66), with the difference being that the client authenticates using a SAML token that is issued by a designated STS. An endorsing token is used to sign the message signature.

In this mechanism, message integrity and confidentiality are protected using ephemeral keys encrypted for the service. Ephemeral keys use an algorithm where the exchange key value is purged from the cryptographic service provider (CSP) when the key handle is destroyed. The service requires messages to be endorsed by a SAML token issued by a designated STS.

Service providers and consumers are in potentially different managed environments. For this mechanism, the service requires that secure communications be endorsed by a trusted STS. The service does not trust the client directly, but instead trusts tokens issued by a designated STS. In other words, the STS is taking on the role of a second service with which the client has to securely authenticate.

For this mechanism, authentication of the client is achieved in this way:

- The client authenticates with the STS and obtains the necessary token with credentials.
- The client's request is signed and encrypted using ephemeral key K.
- The server's response is signed and encrypted using the same K.
- The primary signature of the request is endorsed using the issued token.

To use this mechanism for the web service, you simply select this option as your security mechanism. However, you must have a Security Token Service that can be referenced by the service. An example of an STS can be found in the section Creating and Securing the STS (STS) (page 117). In this section, you select a security mechanism for the STS. The security configuration for the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application. The client truststore must contain the certificate of the STS, which has the alias of `wssip` if you are using the updated GlassFish certificates.

Configuring SSL and Authorized Users

This chapter discusses configuring security for your web service and web service client using the WSIT security mechanisms. Some of these mechanisms require some configuration outside of NetBeans IDE. Depending upon which security mechanism you plan to use, some of the following tasks will need to be completed:

- If you are using the GlassFish container and *message* security, you must update the GlassFish keystore and truststore by importing v3 certificates. The procedure for updating the certificates is described in Updating GlassFish Certificates (page 74).
- If you are using a security mechanism that requires a user to enter a user name and password, create authorized users for your container. Steps for creating an authorized user for the GlassFish container are described in Adding Users to GlassFish (page 72).

- To use a mechanism that uses secure transport (SSL), you must configure the *system* to point to the client and server keystore and truststore files. Steps for doing this are described in Configuring SSL For Your Applications (page 69).

Configuring SSL For Your Applications

This section describes adding the steps to configure your application for SSL. These steps will need to be accomplished for any application that uses one of the mechanisms:

- Transport Security (SSL) (page 62) (see Example: Transport Security (SSL), page 103)
- Message Authentication over SSL (page 64)
- SAML Authorization over SSL (page 64) (see Example: SAML Authorization over SSL (SA), page 106)

The following steps are generic to any application, but have example configurations that will work with the tutorial examples, in particular, Example: SAML Authorization over SSL (SA) (page 106) and Example: Transport Security (SSL) (page 103).

To configure SSL for your application, follow these steps:

1. Select one of the mechanisms that require SSL. These include Transport Security (SSL) (page 62), Message Authentication over SSL (page 64), and SAML Authorization over SSL (page 64).
2. Server Configuration
    - GlassFish is already configured for SSL. No further SSL configuration is necessary if you are using Transport Security. However, if you are using one of the Message Security mechanisms with SSL, you must update the GlassFish certificates as described in Updating GlassFish Certificates (page 74).
    - Configure a user on GlassFish as described in Adding Users to GlassFish (page 72).
3. Client Configuration

   For configuring your system for SSL in order to work through the examples in this tutorial, the same keystore and truststore files are used for both the client and the service. This obviates the needs to set system properties to point to the client stores, as both GlassFish and NetBeans are aware of these certificates and point to them by default.

In general, for the client side of SSL you will not be using the same certificates for the client and the service. In that case, you need to define the client certificate stores by setting the system properties -Djavax.net.ssl.trustStore, -Djavax.net.ssl.keyStore, -Djavax.net.ssl.trustStorePassword, and -Djavax.net.ssl.keyStorePassword in the application client container.

You can specify the environment variables for keystore and truststore by setting the environment variable VMARGS through the shell environment or inside an Ant script, or by passing them in when you start NetBeans IDE from the command line. For example, in the latter case:

```
<NETBEANS_HOME>/bin/netbeans.exe
-J-Djavax.net.ssl.trustStore=
<AS_HOME>/domains/domain1/config/cacerts.jks
-J-Djavax.net.ssl.keyStore=
<AS_HOME>/domains/domain1/config/keystore.jks
-J-Djavax.net.ssl.trustStorePassword=changeit
-J-Djavax.net.ssl.keyStorePassword=changeit
```

Use the hard-coded path to the keystore and truststore files, not variables.

For the SSL mechanism, The browser will prompt you to accept the server alias s1as.

4. On the client side, for the Transport Security (SSL) mechanism, you must either use the fully-qualified hostname in the URL for the service WSDL when you are creating the web sercie client, or else you must follow the steps in Transport Security (SSL) Workaround (page 63).
5. Service Configuration

   To require the service to use the HTTPS protocol, you have to specify the security requirements in the service's application deployment descriptor. This file is ejb-jar.xml for a web service that is implemented as an EJB endpoint, and web.xml for a web service implemented as a servlet. To specify the security information, follow these steps:

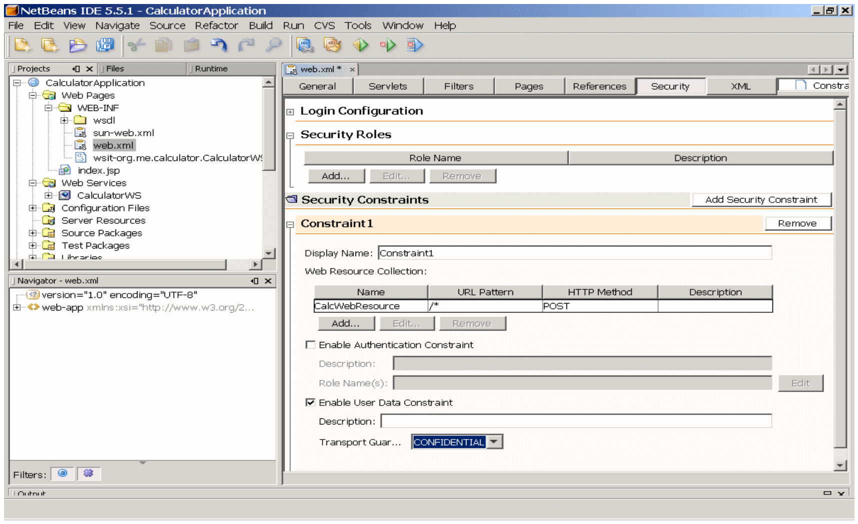
   a. From your web service application expand Web Pages→WEB-INF.
   b. Double-click web.xml (or ejb-jar.xml) to open it in the editor.
   c. Select the Security tab.
   d. On the Security Constraints line, click Add Security Constraint.

e. Under Web Resource Collection, click Add.

f. Enter a Name for the Resource (for example, `CalcWebResource`), and enter the URL Pattern to be protected (for example, `/*`). Select which HTTP Methods to protect, for example, POST. Click OK to close this dialog.

g. Check the Enable User Data Constraint box. Select CONFIDENTIAL for the Transport Guarantee to specify that the application uses SSL because the application requires that data be transmitted so as to prevent other entities from observing the contents of the transmission.

h. The IDE looks like this:



Display Name: Constraint1

Web Resource Collection:

Name	URL Pattern	HTTP Method	Description
CalcWebResource	/*	POST	

☐ Enable Authentication Constraint

Description:

Role Name(s):

☑ Enable User Data Constraint

Description:

Transport Guar...: CONFIDENTIAL

Figure 6–4 Deployment Descriptor page

i. Click the XML tab to display the additions to `web.xml`. The security constraint looks like this:

```
<security-constraint>
   <display-name>Constraint1</display-name>
   <web-resource-collection>
      <web-resource-name>
         CalcWebResource
      </web-resource-name>
      <description/>
      <url-pattern>/*</url-pattern>
      <http-method>POST</http-method>
```

```
    </web-resource-collection>
    <user-data-constraint>
      <description/>
      <transport-guarantee>
        CONFIDENTIAL
      </transport-guarantee>
    </user-data-constraint>
  </security-constraint>
```

j. When you run this project (right-click, select Run Project), the browser will ask you to accept the server certificate of `s1as`. Accept this certificate. The WSDL displays in the browser.

6. Creating a Client

   When creating your client application, use the fully-qualified hostname to specify the secure WSDL location (use **`https://<`*`fully_qualified_hostname`*`>:8181`**`/CalculatorApplication/CalculatorWSService?wsdl`, for example, in place of `http://localhost:8080/CalculatorApplication/CalculatorWSService?wsdl`).

   In some cases, you might get an error dialog telling you that the URL `https://<`*`fully_qualified_hostname`*`>:8181/CalculatorApplication/CalculatorWSService?wsdl` couldn't be downloaded. However, this is the correct URL, and it does load when you run the service. So, when this error occurs, repeat the steps that create the Web Service Client using the secure WSDL. The second time, the web service reference is created and you can continue creating the client.

Adding Users to GlassFish

The following topics are covered:

- Adding Users to GlassFish Using Admin Console (page 72)
- Adding Users to GlassFish From Command Line (page 73)

Adding Users to GlassFish Using Admin Console

To add users to GlassFish using the Admin Console, follow these steps:

1. Start GlassFish if you haven't already done so.

2. Start the Admin Console if you haven't already done so. You can start the Admin Console by starting a web browser and entering the URL `http://localhost:4848/asadmin`. If you changed the default Admin port during installation, enter the correct port number in place of `4848`.
3. To log in to the Admin Console, enter the user name and password of a user in the `admin-realm` who belongs to the `asadmin` group. The name and password entered during installation will work, as will any users added to this realm and group subsequent to installation.
4. Expand the Configuration node in the Admin Console tree.
5. Expand the Security node in the Admin Console tree.
6. Expand the Realms node. Select the `file` realm.
7. Click the Manage Users button.
8. Click New to add a new user to the realm.
9. Enter the correct information into the User ID, Password, and Group(s) fields. The example applications reference a user with the following attributes:
   a. User ID = `wsitUser`
   b. Group List = `wsit`
   c. New Password = `changeit`
   d. Confirm New Password = `changeit`
10. Click OK to add this user to the list of users in the realm.
11. Click Logout when you have completed this task.

Adding Users to GlassFish From Command Line

To add users to GlassFish from the command line, make sure GlassFish is running, then enter the following command:

```
<AS_HOME>/asadmin create-file-user --groups wsit wsitUser
```

Enter `changeit` for the password when prompted.

Configuring Keystores and Truststores

This section describes configuring keystores and truststores. Security mechanisms that use certificates require keystore and truststore files for deployment.

- For GlassFish, default keystore and truststore files come bundled, however WSIT security mechanisms for *message* security require X.509 version 3 certificates. GlassFish contains version 1 certificates, therefore, to enable the WSIT applications to run on GlassFish, you will need to follow the instructions in Updating GlassFish Certificates (page 74).
- For Tomcat, keystore and truststore files do not come bundled with this container, so they must be provided. You can download valid keystore and truststore files for the client and server from `https://xwss.dev.java.net/`.

The following sections discuss how to specify and configure the keystore, truststore, and validators.

Updating GlassFish Certificates

The WSIT message security mechanisms require the use of v3 certificates. The default GlassFish keystore and truststore do not contain v3 certificates at this time (but should before FCS). (GlassFish instances installed using JDK 1.6 do have a v3 certificate but the certificate lacks a particular extension required for supporting some secure WSIT mechanisms.) In order to use message security mechanisms with GlassFish, it is necessary to download keystore and truststore files that contain v3 certificates and import the appropriate certificates into the default GlassFish stores.

To update the GlassFish certificates, follow these steps.

1. Download the zip file that contains the certificates and the Ant scripts (`copyv3.zip`) by going to this URL: `https://xwss.dev.java.net/servlets/ProjectDocumentList?folderID=6645&expandFolder=6645&folderID=6645`.
2. Unzip this file and change into its directory, `copyv3`.
3. Verify that an environment variable named `AS_HOME` is created, and that it specifies the full path to the location of your GlassFish installation, for example, `C:\Sun\GlassFish`.

   NOTE: Some releases of GlassFish may have different default passwords for the keystores. If you are using a different version of GlassFish than the one recommended at `wsit.dev.java.net`, edit the file `build.xml` and specify the correct default password in the `AS_KEYSTORE_PASSWORD` field.

4. From the `copyv3` directory, execute the Ant command that will copy the keystore and truststore files to the appropriate location, and import the appropriate certificates into the GlassFish keystore and truststore. This Ant command is simply: `<AS_HOME>/lib/ant/bin/ant`

   The command window will echo back the certificates that are being added to the keystore and truststore files, and should look something like this:

```
[echo] WARNING: currently we add non-CA certs to GF trust-
store, this will not be required in later releases when we
WSIT starts supporting CertStore(s)
     [java] Added Key Entry  :xws-security-server
     [java] Added Key Entry  :xws-security-client
     [java] Added Trusted Entry  :xwss-certificate-authority
     [java] Added Key Entry  :wssip
     [java] Added Trusted Entry  :xws-security-client
     [java] Added Trusted Entry  :xws-security-server
     [java] Added Trusted Entry  :wssip
     [echo] Adding JVM Option for https outbound alias, this
will take atleast One Minute.
...
```

5. If you'd like to verify that the updates were successful, follow these steps:

   a. Change to the directory containing the GlassFish keystore and truststore files, `<AS_HOME>/domains/domain1/config`.

   b. Verify that the v3 certificate has been imported into the GlassFish truststore. To do this, run the following keytool command:

```
<JDK_HOME>/bin/keytool -list -keystore cacerts.jks -alias
wssip -storepass changeit
```

   If the certificates are *successfully* updated, your response will look something like this:

```
wssip, Aug 20, 2007, trustedCertEntry,
Certificate fingerprint (MD5):
1A:0E:E9:69:7D:D0:80:AD:5C:85:47:91:EB:0D:11:B1
```

   If the certificates were *not* successfully update, your response will look something like this:

```
keytool error: java.lang.Exception: Alias <wssip> does not
exist
```

   c. Verify that the v3 certificate has been imported into the GlassFish keystore. To do this, run the following keytool command:

```
<JDK_HOME>/bin/keytool -list -keystore keystore.jks -alias
xws-security-server -storepass changeit
<JDK_HOME>/bin/keytool -list -keystore keystore.jks -alias
xws-security-client -storepass changeit
```

If the certificates were *successfully* updated, your response should look something like this:

```
xws-security-server, Aug 20, 2007, PrivateKeyEntry,
Certificate fingerprint (MD5):
E4:E3:A9:02:3C:B0:36:0C:C1:48:6E:0E:3E:5C:5E:84
```

If your certificates were *not* successfully update, your response will look more like this:

```
keytool error: java.lang.Exception: Alias <xws-security-
server> does not exist
```

NOTE: The XWSS keystore(s) are *sample* keystores containing sample v3 certificates. These sample keystores can be used for development and testing of security with WSIT technology. Once an application is in production, you should definitely use your own v3 certificates issued by a trusted authority. In order to use WSIT security on GlassFish, you will have to import your trusted stores into GlassFish's keystore and specify those certificates from NetBeans IDE.

Specifying Aliases with the Updated Stores

The configuration of the aliases for all containers (Tomcat, GlassFish) and for all applications (JSR-109-compliant and non-JSR-109-compliant), except for applications that use a Security Token Service (STS) mechanism, is as shown in Table 6–3:

Table 6–3 Keystore and Truststore Aliases

	Keystore Alias	**Truststore Alias**
Client-Side Configuration	xws-security-client	xws-security-server
Server-Side Configuration	xws-security-server	xws-security-client

The configuration of the aliases for applications that use a Security Token Service (STS) mechanism is as shown in Table 6–4:

Table 6–4 Keystore and Truststore Aliases for STS

	Keystore Alias	**Truststore Alias**
Client-Side Configuration	xws-security-client	xws-security-server
STS Configuration	xws-security-client	wssip

Configuring the Keystore and Truststore

NetBeans IDE already knows the location of the default keystore file and its password, but you need to specify which alias is to be used. The following sections discuss configuring the keystore on the service and on the client.

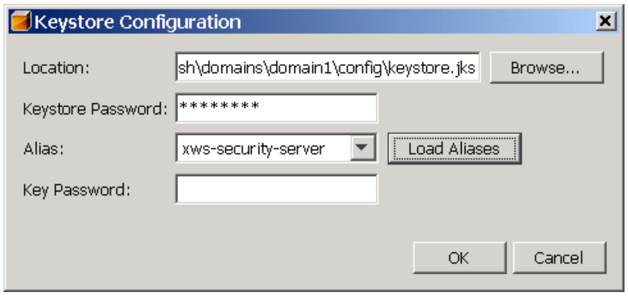
Configuring the Keystore on a Service

A keystore is a database of private keys and their associated X.509 certificate chains authenticating the corresponding public keys. A key is a piece of information that controls the operation of a cryptographic algorithm. For example, in encryption, a key specifies the particular transformation of plaintext into ciphertext, or vice versa during decryption. Keys are used in digital signatures for authentication.

To configure a keystore on a service, perform the following steps:

1. Check the table in Summary of Service-Side Configuration Requirements (page 52) to see if a keystore needs to be configured for the selected security mechanism. If so, continue.
2. Right-click the web service and select Edit Web Service Attributes. The Web Service Attributes editor is displayed.
3. Enable Secure Service, then select a security mechanism.
4. Check the tables in Summary of Service-Side Configuration Requirements (page 52) to see what keystore configuration, if any, is required for that mechanism.

5. Click the Keystore button. The following dialog displays:



Location: sh\domains\domain1\config\keystore.jks
Keystore Password: ********
Alias: xws-security-server
Key Password:

Figure 6–5 Keystore Configuration dialog

6. Depending on what is required for the selected mechanism, you may specify the following information in the Keystore Configuration dialog:
    - **Location**—Use the Browse button to specify the location and name of the keystore. By default, this field specifies the GlassFish keystore file, `<AS_HOME>/domains/domain1/config/keystore.jks`.
    - **Keystore Password**—Specifies the password for the keystore file. If you are running under GlassFish, GlassFish's password is already entered. If you have changed the keystore's password from the default, you must specify the correct value in this field.
    - **Load Aliases**—Click the Load Aliases button to populate the Alias field with the aliases contained in the keystore file. The Location and Store Password fields must be specified correctly for this option to work.
    - **Alias**—Specifies the alias of the certificate in the specified keystore to be used for authentication. Refer to the table in Specifying Aliases with the Updated Stores (page 76) to determine which alias to choose for the selected security mechanism.
    - **Key Password**—Specifies the password of the key within the keystore. For this sample, leave this blank. For this field, the default assumes the key password is the same as the store password, so you only need to specify this field when the key password is different.

    NOTE: The Key Password field enables you to specify a password for the keystore used by the application. When specified, this password is stored in a WSIT configuration file in clear text, which is a security risk. Setting the keystore password in the development environment is fine, however, when you go into production, remember to use the container's Callback Handler to obtain the keys from the keystore. This eliminates the need for the keystore passwords to be supplied by the users. You can also specify the passwords for keystores and truststores by specifying a Callback Handler class that implements the `javax.security.auth.callback.CallbackHandler` interface in the Key Password or Store Password fields.

When creating JSR-109-compliant application, GlassFish will only use the default CallbackHandlers and Validators, and you cannot override the location of the keystore and truststore files. Any attempt to override the default location will be ignored. You do, however, need to specify the keystore and truststore locations in these dialogs in order to specify the alias.

When creating non-JSR-109-compliant application, you can specify the passwords for keystores and truststores by specifying a `CallbackHandler` class that implements the `javax.security.auth.callback.CallbackHandler` interface in the Key Password or Store Password fields.

7. Click OK to close the dialog.

Configuring the Truststore on a Service

A truststore is a database of trusted entities and their associated X.509 certificate chains authenticating the corresponding public keys.

The truststore contains the Certificate Authority (CA) certificates and the certificate(s) of the other party to which this entity intends to send encrypted (confidential) data. This file must contain the public key certificates of the CA and the client's public key certificate. Any kind of encryption without WS-SecureConversation will generally require that a truststore be configured on the *client* side. Any kind of signature without WS-SecureConversation will generally require a truststore on the *server* side.

NOTE: For this release, we are showing that you place the trusted certificates of other parties in GlassFish's truststore, `cacerts.jks`. This is not a recommended practice because any certificate you add to the `cacerts.jks` file effectively means it can be a trusted root for any and all certificate chains, which can be a security problem. In future releases, trusted certificates from other parties will be placed in a certstore and only trusted roots will be placed inside `cacerts.jks`.

To set the truststore configuration options on a service, perform the following steps:

1. Check the table in Summary of Service-Side Configuration Requirements (page 52) to see if a truststore is required for the selected security mechanism. If so, continue.
2. Right-click the web service and select Edit Web Service Attributes. The Web Service Attributes editor is displayed.
3. Enable Secure Service.
4. Click the Truststore button.
5. On the Truststore Configuration page, specify the following options:
   - **Location**—By default, the location and name of the truststore that stores the public key certificates of the CA and the client's public key certificate is already entered. The GlassFish truststore file is `<`*`AS_HOME`*`>/domains/domain1/config/cacerts.jks`.
   - **Store Password**—Specifies the password for the truststore. If you are using GlassFish, the value of `changeit` is already entered. If you have changed the value of the truststore password, you must enter the new value in this field.

     NOTE: The Store Password field enables you to specify a password for the truststore used by the application. When specified, this password is stored in a WSIT configuration file in clear text, which is a security risk. Setting the truststore password in the development environment is fine, however, when you go into production, remember to use the container's Callback Handler to obtain the keys from the truststore. This eliminates the need for the truststore passwords to be supplied by the users. You can also specify the passwords for keystores and truststores by specifying a `CallbackHandler` class that implements the `javax.security.auth.callback.CallbackHandler` interface in the Key Password or Store Password fields.

     When creating JSR-109-compliant application, GlassFish will only use the default CallbackHandlers and Validators, and you cannot override the location of the keystore and truststore files. Any attempt to override the default location will be ignored. You do, however, need to specify the keystore and truststore locations in these dialogs in order to specify the alias.

- **Load Aliases**—Click the Load Aliases button to populate the Alias field with the aliases contained in the truststore file. The Location and Store Password fields must be specified correctly for this option to work.
- **Alias**—Specifies the peer alias of the certificate in the truststore that is to be used when the client needs to send encrypted data. Refer to the table in Specifying Aliases with the Updated Stores (page 76) to determine which alias is appropriate for the selected security mechanism.

  A truststore contains trusted other-party certificates and certificates of Certificate Authorities (CA). A peer alias is the alias of the other party (peer) that the sending party needs to use to encrypt the request.

6. Click OK to close the dialog.

Configuring the Keystore and Truststore on a Client

On the client side, a keystore and truststore file must be configured for some of the security mechanisms. Refer to the table in Summary of Client-Side Configuration Requirements (page 54) for information on which mechanisms require the configuration of keystores and truststores. If the mechanism configured for the service requires the configuration of keystores and truststores, follow these steps:

1. Check the table in Summary of Client-Side Configuration Requirements (page 54) to see if a keystore needs to be configured for the client for the selected security mechanism. If so, continue.
2. In the Projects window, expand the node for the web service client.
3. Expand the Web Service References node.
4. Right-click the node for the web service reference for which you want to configure security options.
5. Select Edit Web Service Attributes.

   When the Web Service References Attributes Editor is opened, select the WSIT Configuration tab to display the WSIT options.

6. Expand the Certificates section to specify the keystore and truststore information if required by the service.
7. Depending on what is required for the selected mechanism, you may specify the following information in the Certificates section:

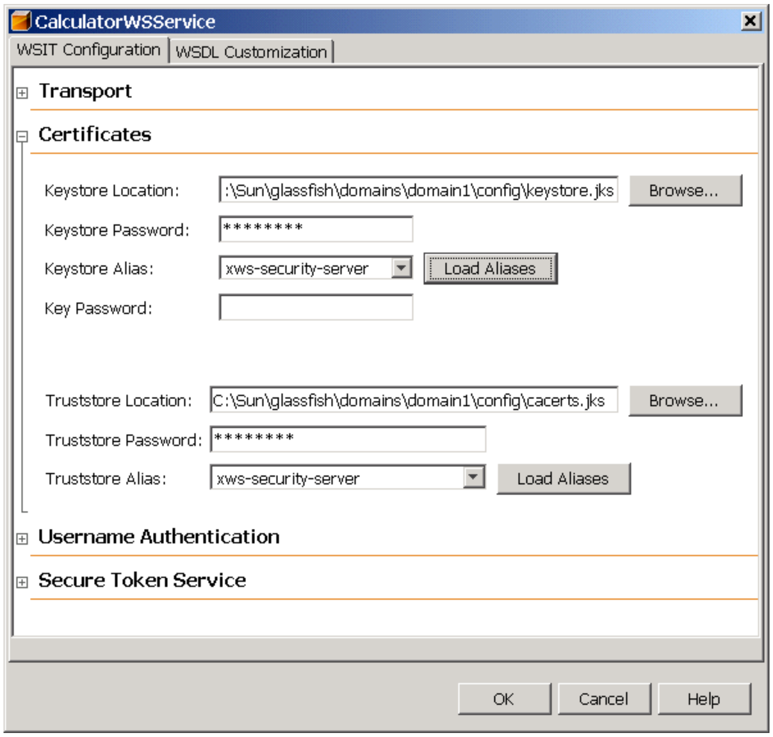
- **Keystore Location**—The directory and file name containing the certificate key to be used to authenticate the client. By default, the location is already set to the default GlassFish keystore, `<AS_HOME>/domains/domain1/config/keystore.jks`
- **Keystore Password**—The password for the keystore used by the client. By default, the password for the GlassFish keystore is already entered. This password is `changeit`.

  NOTE: When specified, this password is stored in a WSIT configuration file in clear text. Setting the keystore password in the development environment is fine, however, when you go into production, remember to use the container's default `CallbackHandler` to obtain the keys from the keystore. This eliminates the need for the keystore passwords to be supplied by the users. You can also specify the passwords for keystores and truststores by specifying a `CallbackHandler` class that implements the `javax.security.auth.callback.CallbackHandler` interface in the Keystore Password, Truststore Password, or Key Password fields.
- **Load Aliases**—Click this button to populate the Alias list with all of the certificates available in the selected keystore. This option will only work if the keystore location and password are correct.
- **Keystore Alias**—Select the alias in the keystore. Refer to the table in Specifying Aliases with the Updated Stores (page 76) to determine which alias is appropriate for the selected security mechanism.
- **Key Password**—If the client key has been password-protected, enter the password for this key. The GlassFish certificate key password is `changeit`.
- **Truststore Location**—The directory and file name of the client truststore containing the certificate of the server. By default, this field points to the default GlassFish truststore, `<AS_HOME>/domains/domain1/config/cacerts.jks`.
- **Truststore Password**—The password for the truststore used by the client. By default, the password for the GlassFish truststore is already specified. The password is `changeit`.

  NOTE: When specified, this password is stored in a WSIT configuration file in clear text. Setting the truststore password in the development environment is fine; however, when you go into production, remember to use the container's default `CallbackHandler` to obtain the keys from the keystore. This eliminates the need for the keystore passwords to be supplied by the users. You can also specify the passwords for keystores and truststores by specifying a `CallbackHandler` class that implements the `javax.security.auth.callback.CallbackHandler` interface in the Keystore Password, Truststore Password, or Key Password fields.

- **Load Aliases**—Click this button to populate the Alias list with all of the certificates available in the selected keystore. This option will only work if the truststore location and password are correct.
- **Truststore Alias**—Select the alias of the server certificate and private key in the client truststore. Refer to the table in Specifying Aliases with the Updated Stores (page 76) to determine which alias is appropriate for the selected security mechanism.

8. When the certificates are configured as suggested for some of the examples in this chapter, the dialog will look like this:



Keystore Location: :\Sun\glassfish\domains\domain1\config\keystore.jks
Keystore Password: ********
Keystore Alias: xws-security-server
Key Password:
Truststore Location: C:\Sun\glassfish\domains\domain1\config\cacerts.jks
Truststore Password: ********
Truststore Alias: xws-security-server

Figure 6–6 Client-side Certificate Configuration Dialog

9. Click OK to close the dialog.

Configuring Validators

A validator is an optional set of classes used to check the validity of a token, a certificate, a timestamp, or a username and password.

Applications that run under a GlassFish 9.1 container do not need to configure Callback Handlers and Validators when using the IDE with WSIT enabled. This is because the container handles the callbacks and validation. You only need to make sure that the certificates are available at locations that GlassFish requires and/or create authorized users using the Admin Console (described in Adding Users to GlassFish (page 72).

Validators are always optional because there are defaults in the runtime (regardless of the container and regardless of whether the application is a JSR-109 or a non-JSR-109 deployment.) For non-JSR-109 deployment, you only need to specify a validator when you want to override the default validators. For JSR-109 deployments, there is no point in specifying an overriding validator, as these will be overridden back to the defaults by GlassFish, thus the Validators button is not available when the selected web service is a JSR-109-compliant application.

To set the validator configuration options for a non-JSR-109-compliant application (such as a J2SE client), perform the following steps:

1. Right-click the web service and select Edit Web Service Attributes. The Web Service Attributes editor is displayed.
2. Enable Secure Service.
3. Click the Validator button.
4. On the Validator Configuration page, specify the following options, when necessary:
   - **Username Validator**—Specifies the validator class to be used to validate username and password on the server side. This option is only used by a web service.

     NOTE: When using the default Username Validator, make sure that the username and password of the client are registered with GlassFish (using Admin Console, described in Adding Users to GlassFish, page 72) if using GlassFish, or is included in the `tomcat-users.xml` file if using Tomcat.
   - **Timestamp Validator**—Specifies the validator class to be used to check the token timestamp to determine whether the token has expired or is still valid.

- **Certificate Validator**—Specifies the validator class to be used to validate the certificate supplied by the client or the web service.
- **SAML Validator**—Specifies the validator class to be used to validate SAML token supplied by the client or the web service.

5. Click OK to close the dialog.

Securing an Operation

This section discusses specifying security mechanisms at the level of a web service operation.

The <*operation_name*>Operation section consists of three subsections. These include:

- **Operation**

  At times, you may need to configure different operations with different supporting tokens. You may wish to configure security at the operation level, for example, in the situation where only one operation requires a UsernameToken to be passed and the rest of the operations do not require this, or in the situation where only one operation needs to be endorsed by a special token and the others do not.

- **Input Message and Output Message**

  Security mechanisms at this level are used to specify what is being protected and the level of protection required.

  In this section, you can specify parts of a message that require integrity protection (digital signature) and/or confidentiality (encryption). When you do this, the specified part of the message, outside of security headers, requires signature and/or encryption. For example, a message producer might submit an order that contains an `orderID` header. The producer signs and/or encrypts the `orderID` header (the SOAP message header) and the body of the request (the SOAP message body). Parts that can be signed and/or encrypted include the body, the header, the local name of the SOAP header, and the namespace of the SOAP header.

  You can also specify arbitrary elements in the message that require integrity protection and/or confidentiality. Because of the mutability of some SOAP headers, a message producer may decide not to sign and/or encrypt the SOAP message header or body as a whole, but instead sign and/or encrypt elements within the header and body. Elements that can be signed and/or encrypted include an XPath expression or a URI which indicates the version of XPath to use.

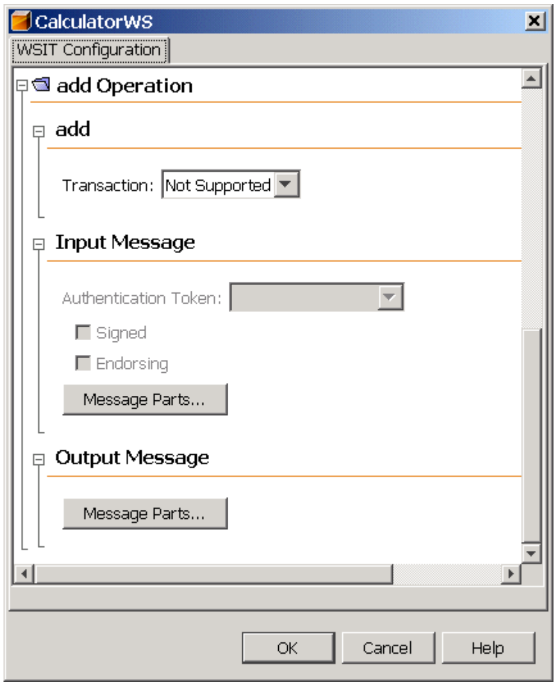
Specifying Security at the Operation, Input Message, or Output Message Level

To specify security mechanisms at the level of the operation, input message, or output message, perform the following steps:

1. Right-click the web service and select Web Service Attributes. The Web Service Attributes editor is displayed.
2. Select Secure Service.
3. Select a security mechanism.

   The following mechanisms do not support Input message level protection:

   - Username Authentication with Symmetric Keys (page 61)
   - Transport Security (SSL) (page 62)
   - Message Authentication over SSL (page 64)
   - SAML Authorization over SSL (page 64)
   - SAML Sender Vouches with Certificates (page 65)
4. Expand the *<operation>*Operation node.
5. Expand the Operation node. It should look like Figure 6–7:



Transaction: Not Supported

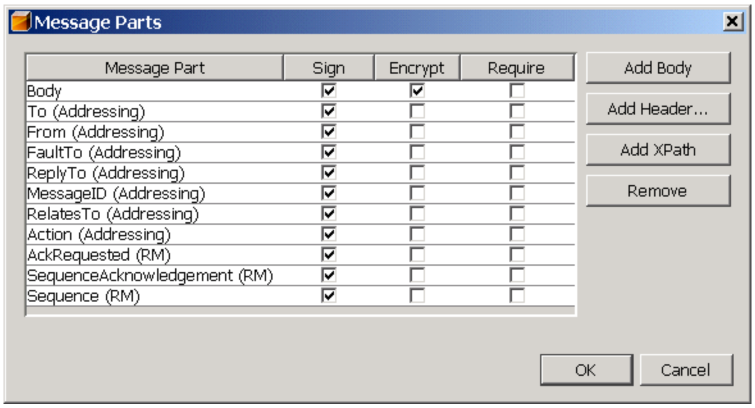
Figure 6–7 Web Service Attributes Editor Page - Operation Level

6. Expand the Operation section. The section will be grayed out if Secure Service is not selected.
7. Specify the following option, as appropriate:
   - **Transactions**—Select an option from the Transactions list to specify a level at which transactions will be secured. For this release, transactions will only use SSL for security. Transactions are discussed in Chapter 10, Using Atomic Transactions.
8. Expand the Input Message section. This section will be grayed out if Secure Service is not selected.
9. Specify the following options, as appropriate:
   - **Authentication Token**— Specifies which supporting token will be used to sign and/or encrypt the specified message parts. Options include Username, X509, SAML, Issued, or None. For further description of these options, read Supporting Token Options (page 89).

- **Signed**—Specifies that the authentication token must be a signed, supporting token. A signed, supporting token is signed by the primary signature token and is part of primary signature.
- **Endorsing**—Specifies that the authentication token must be endorsed. With an endorsing supporting token, the key represented by the token is used to endorse/sign the primary message signature.

If both Signed and Endorsing are selected, the authentication token must be a signed, endorsing, supporting token. In this situation, the token is signed by the primary signature. The key represented by the token is used to endorse/sign the primary message signature.

10.For the Input Message and/or Output Message, click the **Message Parts** button to specify which parts of the message need to be encrypted, signed, and/or required. See the following section for more information on the options in the Message Parts dialog. The Message Parts dialogue displays. It should look like Figure 6–8:



Message Part	Sign	Encrypt	Require
Body	☑	☑	☐
To (Addressing)	☑	☐	☐
From (Addressing)	☑	☐	☐
FaultTo (Addressing)	☑	☐	☐
ReplyTo (Addressing)	☑	☐	☐
MessageID (Addressing)	☑	☐	☐
RelatesTo (Addressing)	☑	☐	☐
Action (Addressing)	☑	☐	☐
AckRequested (RM)	☑	☐	☐
SequenceAcknowledgement (RM)	☑	☐	☐
Sequence (RM)	☑	☐	☐

Figure 6–8 Web Service Attributes Editor Page - Operation Level

11.Click in a checkbox to the right of the message part or element that you would like to sign, encrypt or require.

- Select Sign to specify the parts or elements of a message that require integrity protection (digital signature).
- Select Encrypt to specify the parts or elements of a message that require confidentiality (encryption).
- Select Require to specify the set of parts and/or elements that a message must contain.

12. Click Add Body to add a row for the message body. This will only be necessary if the row has been removed.
13. Click Add Header to add a row for either a specific SOAP header part or for all SOAP header parts. This will only be necessary if the SOAP header row in question has been deleted. The header parts that are available to sign and/or encrypt before clicking the Add Header button include To (Addressing), From (Addressing), FaultTo (Addressing), ReplyTo (Addressing), MessageID (Addressing), RelatesTo (Addressing), and Action (Addressing). After clicking Add Header, and then clicking All Headers, you may also specify AckRequested (RM), SequenceAcknowledgement (RM), and Sequence (RM).
14. There are no XPath elements displayed by default. Click Add XPath to add rows that enable you to specify signature and/or encryption for an XPath expression or a URI which indicates the version of XPath to use. By default, the Required field is selected. This is an editable field. Double click on the XPath row to specify the XPath expression or URI. Only one XPath element is allowed.

    NOTE: There is a limitation when specifying XPath elements. To use XPath elements, switch off Optimize Security manually by adding the `disableStreamingSecurity` policy assertion. For information on how to do this, refer to `http://blogs.sun.com/venu/` for more information on `disableStreamingSecurity`.
15. To remove an element, select it in the Message Part section, and then click Remove to remove it from message security.
16. Click OK to save these settings.

Supporting Token Options

The following are choices for supporting tokens:

- *Username Token*

  A username token is used to identify the requestor by their username, and optionally using a password (or shared secret, or password equivalent) to authenticate that identity. When using a username token, the user must be configured on GlassFish. For information on configuring users on GlassFish, read Adding Users to GlassFish (page 72).

- *X.509 Certificate*

  An X.509 certificate specifies a binding between a public key and a set of attributes that includes (at least) a subject name, issuer name, serial number, and validity interval. An X.509 certificate may be used to validate a public key that may be used to authenticate a SOAP message or to identify the public key with a SOAP message that has been encrypted. When this option is selected, you must specify a truststore. For information on specifying a truststore, read Configuring the Truststore on a Service (page 79).

- *Issued Token*

  An issued token is a token issued by a trusted Secure Token Service (STS). The service does not trust the client directly, but instead trusts tokens issued by a designated STS. In other words, the STS is taking on the role of a second service with which the client has to securely authenticate. The issued tokens contain a key, which is encrypted for the server and which is used for deriving new keys for signing and encrypting.

- *SAML Token*

  A SAML Token uses Security Assertion Markup Language (SAML) assertions as security tokens.

Configuring A Secure Token Service (STS)

A Secure Token Service (STS) is a Web service that issues security tokens. That is, it makes assertions based on evidence that it trusts, to whoever trusts it (or to specific recipients). To communicate trust, a service requires proof, such as a signature, to prove knowledge of a security token or set of security tokens. A service itself can generate tokens or it can rely on a separate STS to issue a security token with its own trust statement (note that for some security token formats this can just be a re-issuance or co-signature). This forms the basis of trust brokering.

The issued token security model includes a target server, a client, and a trusted third party called a Security Token Service (STS). Policy flows from server to client, and from STS to client. Policy may be embedded inside an issued token assertion, or acquired out-of-hand. There may be an explicit trust relationship between the server and the STS. There must be a trust relationship between the client and the STS.

When the web service being referenced by the client uses any of the STS security mechanisms (refer to table in Summary of Service-Side Configuration Requirements (page 52), an STS must be specified. You can specify the STS in one of these ways.

- On the service side, specify the endpoint of the Issuer element and/or specify the Issuer Metadata Exchange (Mex) address of the STS.

  If you need to create a third-party STS, follow the steps in Creating a Third-Party STS (page 91).

  If you already have an STS that you want to use, follow the steps in Specifying an STS on the Service Side (page 94).

  An example that creates and uses an STS can be found in Example: STS Issued Token (STS) (page 115).
- On the client side, specify the information for a preconfigured STS. This is mainly used for a local STS that is in the same domain as the client. Configuring the STS for the client is described in Specifying an STS on the Client Side (page 94).

Creating a Third-Party STS

Use the STS wizard to create an STS from a WSDL file. When using the STS wizard, provide the name of the STS implementation class. This class must extend `com.sun.xml.ws.security.trust.sts.BaseSTSImpl`. After completing the steps of the wizard, your application will contain a new service that is an STS and includes a provider implementation class, STS WSDL, and a WSIT configuration file with a predefined set of policies.

To use the STS wizard to create an STS, follow these steps:

1. Create a new project for the STS by selecting File→New Project.
2. Select Web, then Web Application, then Next.
3. Enter a Project Name. Click Finish.
4. Right-click the STS Project node, select New, then click File/Folder at the top.
5. Select Web Service from the Categories list.
6. Select Secure Token Service (STS) from the File Type(s) list.
7. Click Next.
8. Enter a name for the Web Service Class Name.

9. Enter or select a name for the Package list.
10. Click Finish.

    The IDE takes a while to create the STS. When created, it displays under the project's Web Services node as *<your_STS>*Service, and the Java file displays in the right pane.

11. The STS wizard creates an empty implementation of provider class. Implement the provider implementation class. An example of this can be found in Creating and Securing the STS (STS) (page 117).
12. Back in the Projects window, right-click the STS project folder, and select Edit Web Service Attributes to configure the STS.
13. Select Secure Service.
14. Select a Security Mechanism (but not one of the STS mechanisms). The example application uses Username Authentication with Symmetric Keys.
15. Select the Configure button. For Algorithm Suite option, specify a value that matches the value of the web service. Set the Key Size to 128 if you have not configured Unlimited Strength Encryption. Select OK to close the configuration dialog.

    NOTE: Some of the algorithm suite settings require that Unlimited Strength Encryption be configured in the Java Runtime Environment (JRE), particularly the algorithm suites that use 256 bit encryption. Instructions for downloading and configuring unlimited strength encryption can be found at the following URLS:

    ```
    http://java.sun.com/products/jce/javase.html
    http://java.sun.com/javase/downloads/index_jdk5.jsp#docs
    ```

16. Select Act as Secure Token Service (STS). The default values will create a valid STS. Optionally, you can change the following configuration options:
    - **Issuer**—Specify an identifier for the issuer for the issued token. This value can be any string that uniquely identifies the STS, for example, `MySTS`.
    - **Contract Implementation Class**—Specify the actual implementation class for the `WSTrustContract` interface that will handle token issuance, validation, etc. Default value is `com.sun.xml.ws.trust.impl.IssueSamlTokenContractImpl` for issuing SAML assertions, or click Browse to browse to another contract implementation class.

- **Life Time of Issued Tokens**—The life span of the token issued by the STS. Default value is 300,000 ms.
- **Encrypt Issued Key**—Select this option if the issued key should be encrypted using the service certificate. Default is true.
- **Encrypt Issued Token**—Select this option if the issued token should be encrypted using the service certificate. Default is false.

17.Optionally, to add one or more Service Providers that have a trust relationship with the STS, click the Add button and specify the following configuration options:

- **Provider Endpoint URI**—The endpoint URI of the service provider.
- **Certificate Alias**—The alias of the certificate of the service provider in the keystore.
- **Token Type**—The type of token the service provider requires, for example, `urn:oasis:names:tc:SAML1.0:assertion`.
- **Key Type**—The type of key the service provider requires. The choices are public key or symmetric key. Symmetric key cryptography relies on a shared secret and is usually faster than public key cryptography. Public key cryptography relies on a key that is made public to all and is primarily used for encryption but can be used for verifying signatures.

18.Click OK to close the Select STS Service Provider dialog.

19.Click OK to close the STS Configuration dialog.

20.Click the Keystore button to configure the keystore. If you are using the updated GlassFish stores, these are the settings:

- Location—Defaults to the location and name of the keystore, *<AS_HOME>*`/domains/domain1/config/keystore.jks`.
- Store Password—Enter or accept `changeit`.
- Load Aliases—Click the Load Aliases button.
- Alias—Select `wssip`.
- Click OK to close the dialog.

21.Right-click the STS Project, select Properties. Select the Run category, and enter the following in the Relative URL field: `/`*<your_STS>*`Service?wsdl`.

22.Run the Project (right-click the Project and select Run Project).

23.To view the STS WSDL, append *<your_STS>*`Service` to the URL of the deployed application in the browser. For the example application (Example: STS Issued Token (STS), page 115), you would view the STS WSDL by browsing to `http://localhost:8080//MySTSProject/MySTSService?wsdl`.

Specifying an STS on the Service Side

This section discusses how to specify a Security Token Service that can be referenced by the service. On the service side, you select a security mechanism that includes STS in its title.

The STS itself is secured using a separate (non-STS) security mechanism. The security configuration of the client-side of this application is dependent upon the security mechanism selected for the STS, and not on the security mechanism selected for the application.

To specify an STS for the web service, follow these steps:

1. Right-click the node for the web service you want to secure.
2. Select Edit Web Service Attributes.
3. Select **Secure Service**.
4. Select a **Security Mechanism** that specifies STS from the list.
5. Click **Configure** to specify the STS information.
6. Enter the **Issuer Address** and/or **Issuer Metadata Address**.

   When the Issuer Address and the Metadata values are the same, you only need to enter the Issuer Address. For the example application, the Issuer Address would be `http://localhost:8080/MySTSProject/MySTSService`.
7. Set the **Algorithm Suite** value so that the algorithm suite value of the service matches the algorithm suite value of the STS.

Specifying an STS on the Client Side

Once you've determined whether the an STS is required to be configured on the client side (see Summary of Client-Side Configuration Requirements, page 54), configure the client Secure Token Service options, as follows:

1. In the Projects window, expand the node for the web services client.
2. Expand the Web Service References node.
3. Right-click the node for the web service reference for which you want to configure security options.

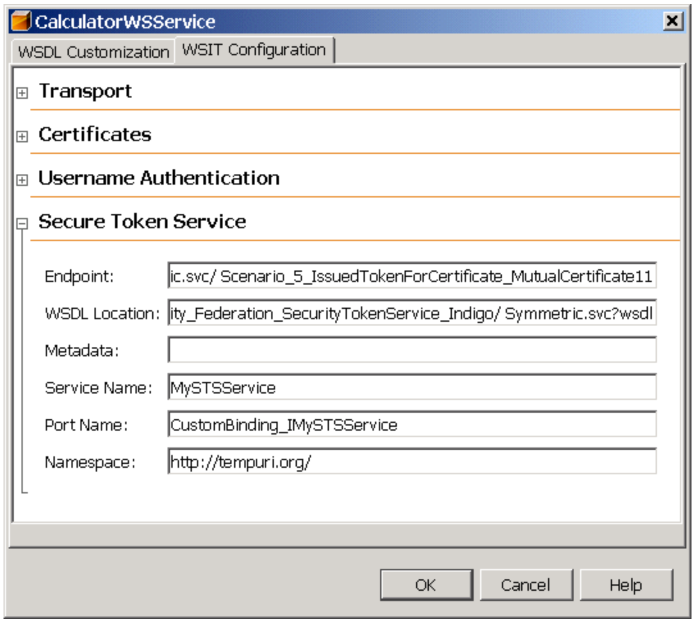
4. Select Edit Web Service Attributes.

   When the Web Service References Attributes Editor is opened, select the WSIT tab to display the WSIT options.

5. Expand the Secure Token Service section to specify the information required by the service. The following options are available for configuration:

   - **Endpoint**—The endpoint of the STS.
   - **WSDL Location**—The location of the WSDL for the STS.
   - **Metadata**—The metadata address for the STS.
   - **Service Name**—The service name of the STS.
   - **Port Name**—The port name of the STS.
   - **Namespace**—The namespace for the service in the WSDL.

   The Endpoint field is a mandatory field. Depending on how you plan to configure the STS, you can provide either Metadata information or information regarding the WSDL Location, Service Name, Port Name and Namespace. The following section contain a few example STS configurations. When the options are configured along the lines of STS Example 2: Endpoint with WSDL Location, Service Name, Port Name, and Namespace (page 96), the dialog looks like this:



Endpoint: ic.svc/ Scenario_5_IssuedTokenForCertificate_MutualCertificate11
WSDL Location: ity_Federation_SecurityTokenService_Indigo/ Symmetric.svc?wsdl
Metadata:
Service Name: MySTSService
Port Name: CustomBinding_IMySTSService
Namespace: http://tempuri.org/

Figure 6–9 WSIT Configuration Page - Secure Token Service on Client

STS Example 1: Endpoint with Metadata

- Endpoint:
  `http://131.107.72.15/`
  `Security_Federation_SecurityTokenService_Indigo/`
  `Symmetric.svc/`
  `Scenario_5_IssuedTokenForCertificate_MutualCertificate11`
- Metadata:
  `http://131.107.72.15//`
  `Security_Federation_SecurityTokenService_Indigo/`
  `Symmetric.svc`

STS Example 2: Endpoint with WSDL Location, Service Name, Port Name, and Namespace

- Endpoint:
  `http://131.107.72.15/`

`Security_Federation_SecurityTokenService_Indigo/`
`Symmetric.svc/`
`Scenario_5_IssuedTokenForCertificate_MutualCertificate11`

- WSDL Location:
  `http://131.107.72.15//`
  `Security_Federation_SecurityTokenService_Indigo/`
  `Symmetric.svc?wsdl`
- Service Name:
  `MySTSService`
- Port Name:
  `CustomBinding_IMySTSService`
- Namespace:
  `http://tempuri.org/`

Example Applications

The following example applications demonstrate configuring web services and web service clients for different security mechanisms. If you are going to work through the examples sequentially, you must manually undo the changes to the service and then refresh the client in order for the client to receive the most recent version of the service's WSDL file, which contains the latest security configuration information.

- Example: Username Authentication with Symmetric Keys (UA) (page 97)
- Example: Mutual Certificates Security (MCS) (page 100)
- Example: Transport Security (SSL) (page 103)
- Example: SAML Authorization over SSL (SA) (page 106)
- Example: SAML Sender Vouches with Certificates (SV) (page 111)
- Example: STS Issued Token (STS) (page 115)

Example: Username Authentication with Symmetric Keys (UA)

The section includes the following topics:

- Securing the Example Service Application (UA) (page 98)
- Securing the Example Web Service Client Application (UA) (page 99)

Securing the Example Service Application (UA)

The following example application starts with the example provided in Chapter 2, WSIT Example Using a Web Container and NetBeans, and demonstrates adding security to both the web service and to the web service client.

For this example, the security mechanism of Username Authentication with Symmetric Keys (page 61) is used to secure the application. To add security to the service part of the example, follow these steps:

1. If you haven't already completed these steps, complete them now:
   a. Update the GlassFish keystore and truststore files as described in Updating GlassFish Certificates (page 74).
   b. Create a user on GlassFish as described in Adding Users to GlassFish (page 72).
2. Create the CalculatorApplication example by following the steps described in the following sections of Chapter 2, WSIT Example Using a Web Container and NetBeans.
   a. Creating a Web Service (page 24)
   b. Skip the section on adding Reliable Messaging.
   c. Deploying and Testing a Web Service (page 28) (first two steps only, do not run the project yet)
3. Expand CalculatorApplication→Web Services, then right-click the node for the web service (CalculatorWS) and select Edit Web Service Attributes.
4. Unselect Reliable Messaging if it is selected.
5. In the CalculatorWSPortBinding section, select Secure Service.
6. From the drop-down list for Security Mechanism, select Username Authentication with Symmetric Keys.
7. Click the Keystore button to provide your keystore with the alias identifying the service certificate. To do this, click the Load Aliases button and select `xws-security-server`. Click OK to close.
8. Click OK to close the WSIT Configuration dialog.

   A new file is added to the project. To view the WSIT configuration file, expand Web Pages→WEB-INF, then double-click the file `wsit-org.me.calculator.CalculatorWS.xml`. This file contains the `sc:KeyStore` element.

9. Right-click the CalculatorApplication node and select Run Project. A browser will open and display the WSDL file for the application.
10. Verify that the WSDL file contains the following elements: `SymmetricBinding` and `UsernameToken`.
11. Follow the steps to secure the client application as described in the next section.

Securing the Example Web Service Client Application (UA)

This section demonstrates adding security to the web service client that references the web service created in the previous section. This web service is secured using the security mechanism described in Username Authentication with Symmetric Keys (page 61). When this security mechanism is used with a web service, the web service client must provide a username and password in addition to specifying the certificate of the server.

To add security to the client that references this web service, complete the following steps:

1. Create the client application by following the steps described in Creating a Client to Consume a WSIT-Enabled Web Service (page 29).

   NOTE: Whenever you make changes on the service, refresh the client so that the client will pick up the change. To refresh the client, right-click the node for the Web Service Reference for the client, and select Refresh Client.

2. Expand the node for the web service client application, CalculatorWSServletClient.
3. Expand the node for Web Service References.
4. Right-click on CalculatorWSService, select Edit Web Service Attributes.
5. Select the WSIT Configuration tab of the CalculatorWSService dialog.
6. For this testing environment, provide a default username and password. To do this,
   a. Expand the Username Authentication node.
   b. Enter the username and password that you created on GlassFish into the Default Username and Default Password fields. If you followed the steps in the section Adding Users to GlassFish (page 72), the user name is `wsitUser` and the password is `changeit`.

NOTE: In a production environment, you should configure a Username Handler and a Password Handler class to eliminate the security risk associated with the default username and password options.

7. Provide the server's certificate by pointing to an alias in the client truststore. To do this, select the Certificates node, click the Load Aliases button, and select `xws-security-server` from the Alias list.
8. Click OK to close this dialog.
9. In the tree, drill down from the project to Source Packages→META-INF. Double-click on CalculatorWSService.xml, and verify that lines similar to the following are present:

```
<wsp:All>
   <wsaws:UsingAddressing xmlns:wsaws=
      "http://www.w3.org/2006/05/addressing/wsdl"/>
   <sc:CallbackHandlerConfiguration
         wspp:visibility="private">
      <sc:CallbackHandler default="wsitUser"
         name="usernameHandler"/>
      <sc:CallbackHandler default="changeit"
         name="passwordHandler"/>
   </sc:CallbackHandlerConfiguration>
   <sc:TrustStore wspp:visibility="private" location=
      "home\glassfish\domains\domain1\config\cacerts.jks"
      storepass="changeit" peeralias="xws-security-server"/>
</wsp:All>
```

10. Right-click on the CalculatorWSServletClient node and select Run Project.

Example: Mutual Certificates Security (MCS)

The section includes the following topics:

- Securing the Example Service Application (MCS) (page 101)
- Securing the Example Web Service Client Application (MCS) (page 102)

Securing the Example Service Application (MCS)

The following example application starts with the example provided in Chapter 2, WSIT Example Using a Web Container and NetBeans, and demonstrates adding security to both the web service and to the web service client.

For this example, the security mechanism of Mutual Certificates Security (page 62) is used to secure the application. To add security to the service part of the example, follow these steps:

1. If you haven't already completed these steps, complete them now:
   a. Update the GlassFish keystore and truststore files as described in Updating GlassFish Certificates (page 74).
2. Create the CalculatorApplication example by following the steps described in the following sections of Chapter 2, WSIT Example Using a Web Container and NetBeans.
   a. Creating a Web Service (page 24)
   b. Skip the section on adding Reliable Messaging.
   c. Deploying and Testing a Web Service (page 28) (first two steps only, do not run the project yet)
3. Expand CalculatorApplication→Web Services, then right-click the node for the web service, CalculatorWS, and select Edit Web Service Attributes.
4. Unselect Reliable Messaging if it is selected.
5. Select Secure Service.
6. From the drop-down list for Security Mechanism, select Mutual Certificates Security.
7. Click the Keystore button, then click the Load Aliases button and select `xws-security-server`. Click OK to close the dialog.
8. Click OK to close the WSIT Configuration dialog.

   A new file is added to the project. To view the WSIT configuration file, expand Web Pages→WEB-INF, then double-click the file `wsit-org.me.calculator.CalculatorWS.xml`. This file contains the `sc:KeyStore` element.
9. Right-click the CalculatorApplication node and select Run Project. A browser will open and display the WSDL file for the application.
10. Verify that the WSDL file contains the `AsymmetricBinding` element.

11. Follow the steps to secure the client application as described in the next section.

Securing the Example Web Service Client Application (MCS)

This section demonstrates adding security to the web service client that references the web service created in the previous section. This web service is secured using the security mechanism described in Mutual Certificates Security (page 62).

To add security to the client that references this web service, complete the following steps:

1. Create the client application following the steps described in Creating a Client to Consume a WSIT-Enabled Web Service (page 29).

   NOTE: Whenever you make changes on the service, refresh the client so that the client will pick up the change. To refresh the client, right-click the node for the Web Service Reference for the client, and select Refresh Client.

2. Expand the node for the web service client, CalculatorWSServletClient.
3. Expand the node for Web Service References.
4. Right-click on CalculatorWSService, select Edit Web Service Attributes.
5. Select the WSIT Configuration tab of the CalculatorWSService dialog.
6. Provide the client's private key by pointing to an alias in the keystore. To do this,
   a. Expand the Certificates node.
   b. Click the Load Aliases button for the keystore.
   c. Select `xws-security-client` from the Alias list.
7. Provide the server's certificate by pointing to an alias in the client truststore. To do this, from the Certificates node,
   a. Click the Load Aliases button for the truststore.
   b. Select `xws-security-server` from the Alias list.
   c. Click OK to close this dialog.
8. In the tree, drill down from the project to Source Packages→META-INF. Double-click on CalculatorWSService.xml, and verify that lines similar to the following are present:

```
<wsp:All>
   <wsaws:UsingAddressing xmlns:wsaws=
      "http://www.w3.org/2006/05/addressing/wsdl"/>
   <sc:KeyStore wspp:visibility="private" location=
      "C:\Sun\glassfish\domains\domain1\config\keystore.jks"
      storepass="changeit" alias="xws-security-server"
      keypass="changeit"/>
   <sc:TrustStore wspp:visibility="private" location=
      "C:\Sun\glassfish\domains\domain1\config\cacerts.jks"
      storepass="changeit"
      peeralias="xws-security-server"/>
</wsp:All>
```

9. Compile and run this application by right-clicking on the Calculator-WSServletClient node and selecting Run Project.

Example: Transport Security (SSL)

This section includes the following topics:

- Securing the Example Service Application (SSL) (page 103)
- Securing the Example Web Service Client Application (SSL) (page 105)

Securing the Example Service Application (SSL)

The following example application starts with the example provided in Chapter 2, WSIT Example Using a Web Container and NetBeans, and demonstrates adding transport security to both the web service and to the web service client.

For this example, the security mechanism of Transport Security (SSL) (page 62) is used to secure the application. To add security to the service part of the example, follow these steps:

1. Create the CalculatorApplication example by following the steps described in the following sections of Chapter 2, WSIT Example Using a Web Container and NetBeans.
   a. Creating a Web Service (page 24)
   b. Skip the section on adding Reliable Messaging.
   c. Deploying and Testing a Web Service (page 28) (first two steps only, do not run the project yet)

2. Expand CalculatorApplication→Web Services, then right-click the node for the web service, CalculatorWS, and select Edit Web Service Attributes.
3. Unselect Reliable Messaging if it is selected.
4. Select Secure Service.
5. From the drop-down list for Security Mechanism, select Transport Security (SSL).
6. Click OK to close the WSIT Configuration dialog.

   A new file is added to the project. To view the WSIT configuration file, expand Web Pages→WEB-INF, then double-click the file `wsit-org.me.calculator.CalculatorWS.xml`.

   NOTE: For Transport Security, the keystore and truststore files are configured outside of the NetBeans UI, in GlassFish. The keystore and truststore files for basic SSL come pre-configured with GlassFish, so there are no additional steps required for this configuration.

7. To require the service to use the HTTPS protocol, you have to specify the security requirements in the service's application deployment descriptor, which is `web.xml` for a web service implemented as a servlet. To specify the security information, follow these steps:
   a. From your web service application expand Web Pages→WEB-INF.
   b. Double-click `web.xml` to open it in the editor.
   c. Select the Security tab.
   d. On the Security Constraints line, click Add Security Constraint.
   e. Under Web Resource Collection, click Add.
   f. Enter a Name for the Resource, `CalcWebResource`. Enter the URL Pattern to be protected, `/*`. Select which HTTP Methods to protect, for example, POST. Click OK to close this dialog.
   g. Check the Enable User Data Constraint box. Select CONFIDENTIAL as the Transport Guarantee to specify that the application uses SSL.
   h. Click the XML tab to view the resulting deployment descriptor additions.
8. Right-click the CalculatorApplication node and select Run Project. If the server presents its certificate, s1as, accept this certificate. A browser will open and display the WSDL file for the application.
9. Follow the steps to secure the client application as described in the next section.

Securing the Example Web Service Client Application (SSL)

This section demonstrates adding security to the web service client that references the web service created in the previous section. This web service is secured using the security mechanism described in Transport Security (SSL) (page 62).

To add security to the client that references this web service, complete the following steps:

1. Create the client application by following the steps described in Creating a Client to Consume a WSIT-Enabled Web Service (page 29), with the exception that you need to specify the secure WSDL when creating the Web Service Client. To do this, create the client application up to the step where you create the Servlet (step 7 as of this writing) by following the steps described in Creating a Client to Consume a WSIT-Enabled Web Service (page 29), with the following exception:
   a. In the step where you are directed to cut and paste the URL of the web service that you want the client to consume into the WSDL URL field, enter `http`**`s://`*`<fully-qualified-hostname>`*`:8181`**`/CalculatorApplication/CalculatorWSService?wsdl` (changes indicated in **bold**) to indicate that this client should reference the web service using the secure port. The first time you access this service, accept the certificate (s1as) when you are prompted. This is the server certificate popping up to confirm its identity to the client.

      In some cases, you might get an error dialog telling you that the URL `https://`*`<fully-qualified-hostname>`*`:8181/CalculatorApplication/CalculatorWSService?wsdl` couldn't be downloaded. However, this the correct URL, and it does load when you run the service. So, when this error occurs, repeat the steps that create the Web Service Client using the secure WSDL. The second time, the web service reference is created and you can continue creating the client.

      NOTE: If you prefer to use `localhost` in place of the fully-qualified hostname (FQHN) in this example, you must follow the steps in Transport Security (SSL) Workaround (page 63).

   b. Continue creating the client following the remainder of the instructions in Creating a Client to Consume a WSIT-Enabled Web Service (page 29).

NOTE: Whenever you make changes on the service, refresh the client so that the client will pick up the change. To refresh the client, right-click the node for the Web Service Reference for the client, and select Refresh Client.

2. Compile and run this application by right-clicking on the CalculatorWSServletClient node and selecting Run Project.

Example: SAML Authorization over SSL (SA)

The section includes the following topics:

- Securing the Example Service Application (SA) (page 106)
- Securing the Example Web Service Client Application (SA) (page 108)

Securing the Example Service Application (SA)

The following example application starts with the example provided in Chapter 2, WSIT Example Using a Web Container and NetBeans, and demonstrates adding security to both the web service and to the web service client.

For this example, the security mechanism of SAML Authorization over SSL (page 64) is used to secure the application. The steps are similar to the ones described in Example: Username Authentication with Symmetric Keys (UA) (page 97), with the addition of the writing of a client-side SAML callback handler to populate the client's request with a SAML assertion.

To add security to the service part of the example, follow these steps:

1. If you haven't already completed these steps, complete them now:
   a. Update the GlassFish keystore and truststore files as described in Updating GlassFish Certificates (page 74).
   b. Create a user on GlassFish as described in Adding Users to GlassFish (page 72).
2. Create the CalculatorApplication example by following the steps described in the following sections of Chapter 2, WSIT Example Using a Web Container and NetBeans.
   a. Creating a Web Service (page 24)

   b. Skip the section on adding Reliable Messaging.
   c. Deploying and Testing a Web Service (page 28) (first two steps only, do not run the project yet)

3. Expand CalculatorApplication→Web Services, then right-click the node for the web service, CalculatorWS, and select Edit Web Service Attributes.
4. Unselect the Reliable Messaging option if it is selected.
5. Select Secure Service.
6. From the drop-down list for Security Mechanism, select SAML Authorization over SSL.
7. Click the Keystore button to provide your keystore with the alias identifying the service certificate and private key. To do this, click the Load Aliases button and select `xws-security-server`. Click OK to close the dialog.
8. For this example, the Truststore information that you need is specified by default, so there is no need to change these settings.
9. Click OK to exit the WSIT Configuration editor.

   A new file is added to the project. To view the WSIT configuration file, expand Web Pages→WEB-INF, then double-click the file `wsit-org.me.calculator.CalculatorWS.xml`. This file contains the `sc:KeyStore` and `sc:Truststore` elements.

10. To require the service to use SSL, you have to specify the security requirements in the service's application deployment descriptor, which is `web.xml` for a web service implemented as a servlet. To specify the security information, follow these steps:
    a. From your web service application expand Web Pages→WEB-INF.
    b. Double-click `web.xml` to open it in the editor.
    c. Select the Security tab.
    d. On the Security Constraints line, click Add Security Constraint.
    e. Under Web Resource Collection, click Add.
    f. Enter a Name for the Resource, `CalcWebResource`. Enter the URL Pattern to be protected, `/*`. Select which HTTP Methods to protect, for example, POST. Click OK to close this dialog.
    g. Check the Enable User Data Constraint box. Select CONFIDENTIAL as the Transport Guarantee to specify that the application uses SSL.
    h. Click the XML tab to view the resulting deployment descriptor additions.

11. Right-click the CalculatorApplication node and select Run Project. Accept the `s1as` certificate if you are prompted to. A browser will open and display the WSDL file for the application.
12. Verify that the WSDL file contains the `TransportBinding` and `Signed-SupportingTokens` element, which in turn contains a `SamlToken` element.
13. Follow the steps to secure the client application as described in the next section.

Securing the Example Web Service Client Application (SA)

This section demonstrates adding security to the web service client that references the web service created in the previous section. This web service is secured using the security mechanism described in SAML Authorization over SSL (page 64).

To add security to the client that references this web service, complete the following steps:

1. For this example, we are using a non-JSR-109-compliant client for variety. To do this, create the client application up to the step where you create the Servlet (step 7 as of this writing) by following the steps described in Creating a Client to Consume a WSIT-Enabled Web Service (page 29), with the following exceptions:
   a. In the step where you are directed to cut and paste the URL of the web service that you want the client to consume into the WSDL URL field, enter `https`**`://<`*`fully-qualified-hostname`*`>:8181`**`/CalculatorApplication/CalculatorWSService?wsdl`, to indicate that this client should reference the web service using the secure port. The first time you access this service, accept the certificate (`s1as`) when you are prompted. This is the server certificate popping up to confirm its identity to the client.

      In some cases, you might get an error dialog telling you that the URL `https://<`*`fully-qualified-hostname`*`>:8181/CalculatorApplication/CalculatorWSService?wsdl` couldn't be downloaded. However, this the correct URL, and it does load when you run the service. So, when this error occurs, repeat the steps that create the Web Service Client using the secure WSDL. The second time, the web service reference is created and you can continue creating the client.

NOTE: If you prefer to use `localhost` in place of the fully-qualified hostname (FQHN) in this example, you must follow the steps in Transport Security (SSL) Workaround (page 63).

b. Name the application CalculatorClient (since it's not a servlet.).

2. Instead of creating a client servlet as is described in Creating a Client to Consume a WSIT-Enabled Web Service (page 29), we are just going to add the web service operation to the generated index.jsp file to create a non-JSR-109 client. To do this,
   a. If the `index.jsp` file is not open in the right pane, double-click it to open it.
   b. Drill down through the Web Service References node until you get to the `add` operation.
   c. Drag the `add` operation to the line immediately following the following line:
      `<h1>JSP Page</h1>`
   d. Edit the values for `i` and `j` if you'd like.
3. Write a `SAMLCallback` handler for the client side to populate a SAML assertion into the client's request to the service. A suggested method for creating the `SAMLCallbackHandler` is shown below:
   a. Right-click on the CalculatorClient node.
   b. Select New→Java Package.
   c. For Package Name, enter `xwss.saml`.
   d. Click Finish.
   e. Drill down from CalculatorClient→Source Packages→xwss.saml.
   f. Right-click on xwss.saml. Select New→File/Folder.
   g. From the Categories list, select Java Classes.
   h. From the File Types list, select Empty Java File.
   i. Click Next.
   j. For Class Name, enter `SamlCallbackHandler`.
   k. Click Finish.
   l. The empty file displays in the IDE.
   m. Download the example file `SamlCallbackHandler.java` from the following URL:
      `https://xwss.dev.java.net/servlets/ProjectDocumentList?folderID=6645&expandFolder=6645&folderID=6645`

n. Open the file in a text editor.

o. Modify the `home` variable to provide the hard-coded path to your GlassFish installation. For example, modify the line:

```
String home = System.getProperty("WSIT_HOME");
```

to

```
String home = "/home/glassfish";
```

p. Copy the contents of this file into the `SamlCallbackHandler.java` window that is displaying in the IDE.

4. Drill down from CalculatorClient→Web Service References.
5. Right-click on CalculatorWSService, select Edit Web Service Attributes.
6. Select the WSIT Configuration tab of the CalculatorWSService dialog.
7. Provide the client's private key by pointing to an alias in the keystore. To do this, expand the Certificates node, click the Load Aliases button for the keystore, and select `xws-security-client` from the Alias list.

   NOTE: If you are using a certificate other than the updated GlassFish certificates described in Updating GlassFish Certificates (page 74), or are otherwise using a different alias for the client's private key alias, correct the private key alias in the line in the `SAMLCallbackHandler.java` file that looks like this:

   ```
   String client_priv_key_alias="xws-security-client";
   ```

   NOTE: If you are using different keystore/truststore files than those described in Updating GlassFish Certificates (page 74), edit the following code in the `SAMLCallbackHandler.java` file accordingly:

   ```
   this.keyStoreURL = home + fileSeparator + "domains" +
   fileSeparator + fileSeparator + "config" + "domain1" +
   fileSeparator + "keystore.jks";
   this.keyStoreType = "JKS";
   this.keyStorePassword = "changeit";
   this.trustStoreURL = home + fileSeparator + "domains" +
   fileSeparator + "domain1" + fileSeparator + "config" +
   fileSeparator + "cacerts.jks";
   this.trustStoreType = "JKS";
   this.trustStorePassword = "changeit";
   ```

8. Provide the server's certificate by pointing to an alias in the client truststore. To do this, from the Certificates node, click the Load Aliases button for the Truststore and select `xws-security-server`.

9. Expand the Username Authentication node. In the SAML Callback Handler field, enter the name of the class written in step 3 above, `xwss.saml.SamlCallbackHandler`.
10. Click OK to close this dialog.
11. In the tree, drill down from the project to Source Packages→META-INF. Double-click on CalculatorWSService.xml, and verify that lines similar to the following are present, where `xwss.saml.SamlCallbackHandler` is the SAML Callback Handler class for the client:

```
<wsp:All>
   <wsaws:UsingAddressing xmlns:wsaws=
      "http://www.w3.org/2006/05/addressing/wsdl"/>
   <sc:CallbackHandlerConfiguration
      wspp:visibility="private">
      <sc:CallbackHandler name="samlHandler"
         classname="xwss.saml.SamlCallbackHandler"/>
   </sc:CallbackHandlerConfiguration>
   <sc:KeyStore wspp:visibility="private" location=
      "<GF_HOME>\domains\domain1\config\keystore.jks"
      storepass="changeit" alias="xws-security-client"
      keypass="changeit"/>
   <sc:TrustStore wspp:visibility="private" location=
      "<GF_HOME>\domains\domain1\config\cacerts.jks"
      storepass="changeit"
      peeralias="xws-security-server"/>
</wsp:All>
```

12. Compile and run this application by right-clicking the CalculatorClient node and selecting Run Project.

Example: SAML Sender Vouches with Certificates (SV)

The topics covered in this section include the following:

- Securing the Example Service Application (SV) (page 112)
- Securing the Example Web Service Client Application (SV) (page 113)

Securing the Example Service Application (SV)

The following example application starts with the example provided in Chapter 2, WSIT Example Using a Web Container and NetBeans, and demonstrates adding security to both the web service and to the web service client.

For this example, the security mechanism of SAML Sender Vouches with Certificates (page 65) is used to secure the application. The steps are similar to the ones described in Example: Username Authentication with Symmetric Keys (UA) (page 97), with the addition of the writing of a client-side SAML callback handler to populate the client's request with a SAML assertion.

To add security to the service part of the example, follow these steps:

1. If you haven't already completed these steps, complete them now:
    a. Update the GlassFish keystore and truststore files as described in Updating GlassFish Certificates (page 74).
    b. Create a user on GlassFish as described in Adding Users to GlassFish (page 72).
2. Create the CalculatorApplication example by following the steps described in the following sections of Chapter 2, WSIT Example Using a Web Container and NetBeans.
    a. Creating a Web Service (page 24)
    b. Skip the section on adding Reliable Messaging.
    c. Deploying and Testing a Web Service (page 28) (first two steps only, do not run the project yet)
3. Expand CalculatorApplication→Web Services, then right-click the node for the web service, CalculatorWS, and select Edit Web Service Attributes.
4. Unselect the Reliable Messaging option if it is selected.
5. Select Secure Service.
6. From the drop-down list for Security Mechanism, select SAML Sender Vouches with Certificates.
7. Click the Keystore button to provide your keystore with the alias identifying the service certificate and private key. To do this, click the Load Aliases button and select `xws-security-server`. Click OK to close the dialog.
8. For this example, the Truststore information that you need is specified by default, so there is no need to change these settings.

9. Click OK to exit the WSIT Configuration editor.

   A new file is added to the project. To view the WSIT configuration file, expand Web Pages→WEB-INF, then double-click the file `wsit-org.me.calculator.CalculatorWS.xml`. This file contains the `sc:KeyStore` and `sc:Truststore` elements.

10. Right-click the CalculatorApplication node and select Run Project. Accept the `s1as` certificate if you are prompted to. A browser will open and display the WSDL file for the application.
11. Verify that the WSDL file contains the `TransportBinding` and `SignedSupportingTokens` element, which in turn contains a `SamlToken` element.
12. Follow the steps to secure the client application as described in the next section.

Securing the Example Web Service Client Application (SV)

This section demonstrates adding security to the web service client that references the web service created in the previous section. This web service is secured using the security mechanism described in SAML Sender Vouches with Certificates (page 65).

To add security to the client that references this web service, complete the following steps:

1. For this example, we are using a non-JSR-109-compliant client. To do this, create the client application up to the step where you create the Servlet (step 7 as of this writing) by following the steps described in Creating a Client to Consume a WSIT-Enabled Web Service (page 29), with one exception: name the application CalculatorClient (since it's not a servlet.).
2. Instead of creating a client servlet as is described in Creating a Client to Consume a WSIT-Enabled Web Service (page 29), we are just going to add the web service operation to the generated index.jsp file to create a non-JSR-109 client. To do this,
   a. If the `index.jsp` file is not open in the right pane, double-click it to open it.
   b. Drill down through the Web Service References node until you get to the `add` operation.
   c. Drag the `add` operation to the line immediately following the following line:

```
<h1>JSP Page</h1>
```

   d. Edit the values for `i` and `j` if you'd like.

3. Write a `SAMLCallback` handler for the client side to populate a SAML assertion into the client's request to the service. A suggested method for creating the `SAMLCallbackHandler` is shown below:
   a. Right-click on the CalculatorClient node.
   b. Select New→Java Package.
   c. For Package Name, enter `xwss.saml`.
   d. Click Finish.
   e. Drill down from CalculatorClient→Source Packages→xwss.saml.
   f. Right-click on xwss.saml. Select New→File/Folder.
   g. From the Categories list, select Java Classes.
   h. From the File Types list, select Empty Java File.
   i. Click Next.
   j. For Class Name, enter `SamlCallbackHandler`.
   k. Click Finish.
   l. The empty file displays in the IDE.
   m. Download the example file `SamlCallbackHandler.java` from the following URL:
   `https://xwss.dev.java.net/servlets/ProjectDocumentList?folderID=6645&expandFolder=6645&folderID=6645`
   n. Open the file in a text editor.
   o. Modify the `home` variable to provide the hard-coded path to your GlassFish installation. For example, modify the line:

   ```
   String home = System.getProperty("WSIT_HOME");
   ```

   to

   ```
   String home = "/home/glassfish";
   ```

   p. Set the subject confirmation method to SV (Sender Vouches). For more information on this topic, read Example SAML Callback Handlers (page 59).
   q. Copy the contents of this file into the `SamlCallbackHandler.java` window that is displaying in the IDE.
4. Drill down from CalculatorClient→Web Service References.
5. Right-click on CalculatorWSService, select Edit Web Service Attributes.

6. Select the WSIT Configuration tab of the CalculatorWSService dialog.
7. Provide the client's private key by pointing to an alias in the keystore. To do this, expand the Certificates node, click the Load Aliases button for the keystore, and select `xws-security-client` from the Alias list.
8. Provide the server's certificate by pointing to an alias in the client truststore. To do this, from the Certificates node, click the Load Aliases button for the Truststore and select `xws-security-server`.
9. Expand the Username Authentication node. In the SAML Callback Handler field, enter the name of the class written in step 3 above, `xwss.saml.SamlCallbackHandler`.
10. Click OK to close this dialog.
11. In the tree, drill down from the project to Source Packages→META-INF. Double-click on CalculatorWSService.xml, and verify that lines similar to the following are present, where `xwss.saml.SamlCallbackHandler` is the SAML Callback Handler class for the client:

```
<wsp:All>
   <wsaws:UsingAddressing xmlns:wsaws=
      "http://www.w3.org/2006/05/addressing/wsdl"/>
   <sc:CallbackHandlerConfiguration
      wspp:visibility="private">
      <sc:CallbackHandler name="samlHandler"
         classname="xwss.saml.SamlCallbackHandler"/>
   </sc:CallbackHandlerConfiguration>
   <sc:KeyStore wspp:visibility="private" location=
      "<GF_HOME>\domains\domain1\config\keystore.jks"
      storepass="changeit" alias="xws-security-client"
      keypass="changeit"/>
   <sc:TrustStore wspp:visibility="private" location=
      "<GF_HOME>\domains\domain1\config\cacerts.jks"
      storepass="changeit"
      peeralias="xws-security-server"/>
</wsp:All>
```

12. Compile and run this application by right-clicking the CalculatorClient node and selecting Run Project.

Example: STS Issued Token (STS)

The topics covered in this section include the following:

- Securing the Example Service Application (STS) (page 116)
- Creating and Securing the STS (STS) (page 117)

- Securing the Example Web Service Client Application (STS) (page 119)

Securing the Example Service Application (STS)

The following example application starts with the example provided in Chapter 2, WSIT Example Using a Web Container and NetBeans, and demonstrates adding security to both the web service and to the web service client.

For this example, the security mechanism of STS Issued Token (page 66) is used to secure the application. The steps are similar to the ones described in Example: Username Authentication with Symmetric Keys (UA) (page 97), with the addition of creating and securing an STS.

To add security to the service part of the example, follow these steps:

1. Create a user on GlassFish if you haven't already done so. (see Adding Users to GlassFish, page 72).
2. Create the CalculatorApplication example by following the steps described in the following sections of Chapter 2, WSIT Example Using a Web Container and NetBeans.
   a. Creating a Web Service (page 24)
   b. Skip the section on adding Reliable Messaging.
   c. Deploying and Testing a Web Service (page 28) (first two steps only, do not run the project yet).
3. Expand CalculatorApplication→Web Services, then right-click the node for the web service, CalculatorWS, and select Edit Web Service Attributes.
4. Unselect the Reliable Messaging option if it is selected.
5. Select Secure Service.
6. From the drop-down list for Security Mechanism, select STS Issued Token.
7. Select the Configure button. For Algorithm Suite, select Basic128 bit. For Key Size, select 128. Select OK to close the configuration dialog (the algorithm suite value of the service must match the algorithm suite value of the STS.)

   NOTE: If you have configured Unlimited Strength Encryption as described in Creating a Third-Party STS (page 91), you can leave the key size at 256. Otherwise, you must set it to 128.

8. Click OK to exit the WSIT Configuration editor.

   A new file is added to the project. To view the WSIT configuration file, expand Web Pages→WEB-INF, then double-click the file `wsit-org.me.calculator.CalculatorWS.xml`.

9. Right-click the CalculatorApplication node and select Run Project. This step compiles the application and deploys it onto GlassFish. A browser will open and display the WSDL file for the application.
10. Follow the steps for creating and securing the Security Token Service as described in the next section.

Creating and Securing the STS (STS)

To create and secure a Security Token Service for this example, follow these steps:

1. Create a new project for the STS by selecting File→New Project.
2. Select Web, then Web Application, then Next.
3. Enter `MySTSProject` for the Project Name. Click Finish.
4. Right-click the `MySTSProject` node, select New, then click File/Folder at the top.
5. Select Web Services from the Categories list.
6. Select Secure Token Service (STS) from the File Type(s) list.
7. Click Next.
8. Enter the name `MySTS` for the Web Service Class Name.
9. Select `org.me.my.sts` from the Package list.
10. Click Finish.

    The IDE takes a while to create the STS. When created, it displays under the project's Web Services node as `MySTSService`, and `MySTS.java` displays in the right pane.

11. The STS wizard creates an empty implementation of provider class. Implement the provider implementation class by copying the following code into the `MySTS.java` file:

    a. Add these import statements to the list of imports:

    ```
    import com.sun.xml.ws.security.trust.sts.BaseSTSImpl;
    import javax.annotation.Resource;
    import javax.xml.ws.Provider;
    import javax.xml.ws.Service;
    ```

```
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.WebServiceProvider;
import javax.xml.transform.Source;
import javax.xml.ws.handler.MessageContext;
```

b. Add the following `Resource` annotation after the line `public class MySTS implements javax.xml.ws.Provider<Source> {`:

```
@Resource protected WebServiceContext context;
```

c. Change the following line of code:

```
public class MySTS implements
   javax.xml.ws.Provider<Source>
```

to:

```
public class MySTS extends BaseSTSImpl implements
   javax.xml.ws.Provider<Source>
```

d. For the `invoke` method, replace the `return null` line with the following return statement:

```
return super.invoke(source);
```

e. Add the following method after the `invoke` method:

```
protected MessageContext getMessageContext() {
   MessageContext msgCtx = context.getMessageContext();
   return msgCtx;
}
```

12. Back in the Projects window, expand the MySTSProject node, then expand the Web Services node. Right-click on the MySTSService[IMySTSService_Port] node, and select Edit Web Service Attributes to configure the STS.
13. Select Secure Service if it's not already selected.
14. Verify that the Security Mechanism of Username Authentication with Symmetric Keys is selected.
15. Select the Configure button. For Algorithm Suite, verify that Basic128 bit is selected (so that it matches the value selected for the service.) For the Key Size, verify that 128 is selected. Select OK to close the configuration dialog.
16. Select Act as Secure Token Service (STS). Click OK to close the Select STS Service Provider dialog.

17. Click the Keystore button to provide your keystore with the alias identifying the service certificate and private key. To do this, click the Load Aliases button and then select `wssip`. Click OK to close the dialog.
18. Click OK to close the WSIT Configuration dialog.

    A new file is added to the project. To view the WSIT configuration file, expand Web Pages→WEB-INF→wsdl→MySTS, then double-click the file `MySTSService.wsdl`. This file contains the `sc:KeyStore` element.
19. Right-click the MySTSProject tab, select Properties. Select the Run category, and enter the following in the Relative URL field: `/MySTSService?wsdl`.
20. Run the Project (right-click the project and select Run Project). The STS WSDL displays in the browser.
21. Follow the steps to secure the client application as described in the next section.

Securing the Example Web Service Client Application (STS)

This section demonstrates adding security to the CalculatorApplication's web service client, which was secured using the security mechanism described in STS Issued Token (page 66).

To add security to the client, complete the following steps:

1. Create the client application by following the steps described in Creating a Client to Consume a WSIT-Enabled Web Service (page 29).

   NOTE: Whenever you make changes on the service, refresh the client so that the client will pick up the change. To refresh the client, right-click the node for the Web Service Reference for the client, and select Refresh Client.
2. Drill down from CalculatorWSServletClient→Web Service References.
3. Right-click on CalculatorWSService, select Edit Web Service Attributes. Select the WSIT Configuration tab.
4. Provide the client's private key by pointing to an alias in the keystore. To do this, expand the Certificates node, click the Load Aliases button for the keystore, and select `xws-security-client` from the Alias list.

5. Provide the service's certificate by pointing to an alias in the client truststore. To do this, from the Certificates node, click the Load Aliases button for the truststore and select `xws-security-server` from the Alias list.
6. Expand the Security Token Service node to provide details for the STS to be used. When the Endpoint and the Metadata values are the same, you only need to enter the Endpoint value. For the Endpoint field, enter the following value: `http://localhost:8080/MySTSProject/MySTSService`.
7. Click OK to close this dialog.
8. The service requires a token to be issued from the STS at `http://localhost:8080/MySTSProject/MySTSService`, with WSDL file `http://localhost:8080/MySTSProject/MySTSService?wsdl`. To do this, follow these steps:
   a. Right-click the CalculatorWSServletClient node and select New→Web Service Client. The New Web Service Client window appears.
   b. Select the WSDL URL option.
   c. Cut and paste the URL of the web service that you want the client to consume into the WSDL URL field. For example, here is the URL for the `MySTS` web service:
      ```
      http://localhost:8080/MySTSProject/MySTSService?wsdl
      ```
   d. Type `org.me.calculator.client.sts` in the Package field, and click Finish. The Projects window displays the new web service client.
9. Drill down from CalculatorWSServletClient→Web Service References.
10. Right-click MySTSService, select Edit Web Service Attributes.
11. Select the WSIT Configuration tab of the MySTSService dialog.
12. Provide the client's private key by pointing to an alias in the keystore. To do this, expand the Certificates node, click the Load Aliases button for the keystore, and select `xws-security-client` from the Alias list.
13. Verify the STS's certificate by pointing to an alias in the client truststore. To do this, from the Certificates node, click the Load Aliases button and select `wssip` from the Alias list.
14. Expand the Username Authentication node and verify that the default user name and password as specified in GlassFish. If you followed the steps in Adding Users to GlassFish (page 72), this will be User Name `wsitUser` and Password `changeit`.
15. Click OK to close this dialog.
16. Compile and run this application by right-clicking the CalculatorWSServletClient project and selecting Run Project.

Example: Other STS Examples

Another STS example application can be found at the following URL:

`https://wsit.dev.java.net/source/browse/wsit/wsit/samples/ws-trust/`

Further Information

For more information on securing web applications using the WSIT technology, visit the Project Tango web site at `https://wsit.dev.java.net/`. On this page, you will find information about the specifications implemented in this product, source code, support information, links to documentation updates, and much more.

Some other sources that contain blogs and/or screencasts about using WSIT include the following:

- *Sun WSIT Bloggers*
  `http://pipes.yahoo.com/pipes/pipe.info?_id=2iWQPSDG2xGT0WC7p2IyXQ`
  `http://planet.sun.com/webservices/group/blogs/`
- *Project Tango: An Overview*
  `https://wsit.dev.java.net/docs/tango-overview.pdf`
- Web Services blog
  `http://blogs.sun.com/arungupta/category/webservices`
- *Manual Web Service Configuration In From Java Case* (and others)
  `http://blogs.sun.com/japod/date/20070226`
- *Develop WSTrust Application Using NetBeans* (and others)
  `http://blog.sun.com/shyamrao/`
- *Security in WSIT* (and others)
  `http://blogs.sun.com/ashutosh/category/Sun`
- WSIT Screencasts
  `https://wsit.dev.java.net/screencasts.html`
- Specifications Implemented by WSIT
  `https://wsit.dev.java.net/specification-links.html`

7

WSIT Example Using a Web Container Without NetBeans

THIS chapter describes how to use the two supported web containers—GlassFish version 2 or Apache Tomcat 5.5—to create, build, and deploy a web service and a client that use the Web Services Interoperability Technologies (WSIT). This chapter also includes examples of the files you must create and the build directories.

To run the examples described in this chapter, download the WSIT samples kits, `wsit-enabled-fromjava.zip` and `wsit-enabled-fromwsdl.zip`, from the following location:

```
https://wsit.dev.java.net/source/browse/wsit/wsit/docs/
howto/
```

You could also use NetBeans IDE to create and deploy WSIT web services and clients. The IDE provides a graphical user interface (GUI) and does many of the manual steps described in this chapter for you, thus reducing the amount of coding and processing required. For an example that creates, builds, and deploys a web service and a web service client using NetBeans IDE, see Chapter 2.

This chapter covers the following topics:

- Environment Configuration Settings (page 124)
- WSIT Configuration and WS-Policy Assertions (page 125)
- Creating a Web Service (page 126)
- Building and Deploying the Web Service (page 131)
- Creating a Web Service Client (page 135)
- Building and Deploying a Client (page 139)
- Running a Web Service Client (page 139)
- Undeploying a Web Service (page 139)

Environment Configuration Settings

Before you can build and run the samples in this tutorial, you need to complete the following tasks:

- Setting the Web Container Listener Port (page 124)
- Setting the Web Container Home Directory (page 125)

Setting the Web Container Listener Port

The Java code and configuration files for the examples used in this tutorial assume that the web container is listening on IP port 8080. Port 8080 is the default listener port for both GlassFish (domain1) and Tomcat. If you have changed the port, you must update the port number in the following files before building and running the examples:

- `wsit-enabled-fromjava/etc/wsit-fromjava.server.AddNumber-sImpl.xml`
- `wsit-enabled-fromjava/etc/custom-schema.xml`
- `wsit-enabled-fromjava/etc/custom-client.xml`
- `wsit-enabled-fromjava/etc/build.properties`
- `wsit-enabled-fromwsdl/etc/custom-client.xml`
- `wsit-enabled-fromwsdl/etc/build.properties`

Setting the Web Container Home Directory

Before building and deploying the web service and its client, the home directory of the web container must be set as an environment variable.

When you are running from the command-line, you should set the appropriate environment variable to the web container's top-level installation directory. This way, you will not have to manually set the environment variable each time you open a new command window. For GlassFish, the `AS_HOME` environment variable should be set to the top-level directory of GlassFish, for example, on Windows: `C:/Sun/glassfish`. For Apache Tomcat, set the `CATALINA_HOME` environment variable to the Tomcat top-level directory.

WSIT Configuration and WS-Policy Assertions

WSIT features are enabled and configured using a mechanism defined by the Web Services Policy Framework (WS-Policy) specification. A web service expresses its requirements and capabilities via policies embedded in the service's WSDL description. A web service consumer, or client, verifies that it can handle the expressed requirements and, optionally, uses server capabilities advertised in policies.

Each individual WSIT technology, such as Reliable Messaging, Addressing, or Secure Conversation, provides a set of policy assertions it can process. Those assertions provide the necessary configuration details to the WSIT run-time to enable proper operation of the WSIT features used by a given web service. The assertions may specify particular configuration settings or rely on default settings that are pre-determined by the specific technology. For instance, in the snippet shown below, the `wsrm:AcknowledgementInterval` and `wsrm:InactivityTimeout` settings are both optional and could be omitted. The following snippet shows WS-Policy assertions for WS-Addressing and WS-Reliable Messaging:

```
<wsp:Policy wsu:Id="AddNumbers_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsaw:UsingAddressing/>
      <wsrm:RMAssertion>
```

```
            <wsrm:InactivityTimeout Milliseconds="600000"/>
            <wsrm:AcknowledgementInterval Milliseconds="200"/>
          </wsrm:RMAssertion>
        </wsp:All>
      </wsp:ExactlyOne>
    </wsp:Policy>
```

This snippet is valid in either a WSIT configuration file (`wsit-`*`<package>`*`.`*`<service>`*`.xml`) or in a Web Services Description Language (WSDL) file. This snippet is from the WSIT configuration file in the example `wsit-enabled-fromjava/etc/wsit-fromjava.server.AddNumbersImpl.xml`.

Creating a Web Service

You can create a web service starting from Java code or starting from a WSDL file. The following sections describe each approach:

- Creating a Web Service From Java (page 126)
- Creating a Web Service From WSDL (page 129)

Creating a Web Service From Java

One way to create a web service application is to start by coding the endpoint in Java. If you are developing your Java web service from scratch or have an existing Java class you wish to expose as a web service, this is the most direct approach.

The *Java API for XML Web Services* (JAX-WS) 2.0, JSR-224, relies heavily on the use of annotations as specified in `A Metadata Facility for the Java Programming Language` (JSR-175) and *Web Services Metadata for the Java Platform* (JSR-181), as well as additional annotations defined by the JAX-WS 2.0 specification.

The web service is written as a normal Java class. Then the class and its exposed methods are annotated with the web service annotations `@WebService` and `@WebMethod`. The following code snippet shows an example:

```
@WebService
public class AddNumbersImpl {
  @WebMethod(action="addNumbers")
  public int addNumbers(int number1, int number2)
       throws AddNumbersException {
        if (number1 < 0 || number2 < 0) {
            throw new AddNumbersException(
                    "Negative number cant be added!",
                    "Numbers: " + number1 + ", " + number2);
        }
        return number1 + number2;
    }
}
```

When developing a web service from scratch or based on an existing Java class, WSIT features are enabled using a configuration file. That file, `wsit-<package>.<service>.xml`, is written in WSDL format. An example configuration file can be found in the accompanying samples:

```
wsit-enabled-fromjava/etc/wsit-fromjava.server.AddNumber-
sImpl.xml
```

The settings in the `wsit-<package>.<service>.xml` file are incorporated dynamically by the WSIT run-time into the WSDL it generates for the web service. So when a client requests the web service's WSDL, the run-time embeds any publicly visible policy assertions contained in the `wsit-<package>.<service>.xml` file into the WSDL. For the example `wsit-fromjava.server.AddNumbersImpl.xml` in the sample discussed in this tutorial, the Addressing and Reliable Messaging assertions are part of the WSDL as seen by the client.

Note: The `wsit-<package>.<service>.xml` file must be in the `WEB-INF` sub-directory of the application's WAR file when it is deployed to the web container. Otherwise, the WSIT run-time environment will not find it.

To create a web service from Java, create the following files:

- These files define the web service and the WSIT configuration for the service, which are discussed in the sections below:
    - Web Service Implementation Java File (page 128)
    - wsit-<package>.<service>.xml File (page 129)
- These files are standard files required for JAX-WS. Examples of these files are provided in the `wsit-enabled-fromjava` sample directory.
    - `AddNumbersException.java`
    - `custom-schema.xml`
    - `sun-jaxws.xml`

    - `web.xml`
- These files are standard in any Ant build environment. Examples of these files are provided in the `wsit-enabled-fromjava` sample directory.
    - `build.xml`
    - `build.properties`

Web Service Implementation Java File

The sample files define a web service that takes two integers, adds them, and returns the result. If one of the integers is negative, an exception is thrown.

The starting point for developing a web service that uses the WSIT technologies is a Java class file annotated with the `javax.jws.WebService` annotation. The `@WebService` annotation defines the class as a web service endpoint.

The following file (`wsit-enabled-fromjava/src/fromjava/serverAddNumbersImpl.java`) implements the web service interface.

```
package fromjava.server;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class AddNumbersImpl {
  @WebMethod(action="addNumbers")
  public int addNumbers(int number1, int number2)
       throws AddNumbersException {
     if (number1 < 0 || number2 < 0) {
       throw new AddNumbersException(
            "Negative number cannot be added!",
            "Numbers: " + number1 + ", " + number2);
     }
     return number1 + number2;
  }
}
```

Note: To ensure interoperability with Windows Communication Foundation (WCF) clients, you must specify the `action` element of `@WebMethod` in your endpoint implementation classes. WCF clients will incorrectly generate an empty string for the Action header if you do not specify the `action` element.

wsit-<*package*>.<*service*>.xml File

This file is the WSIT configuration file. It defines which WSIT technologies are enabled in the web service. The snippet shown below illustrates how to enable the WSIT reliable messaging technology in a `wsit-<package>.<service>.xml` file.

```
<wsp:Policy wsu:Id="AddNumbers_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsaw:UsingAddressing/>
      <wsrm:RMAssertion>
        <wsrm:InactivityTimeout Milliseconds="600000"/>
        <wsrm:AcknowledgementInterval Milliseconds="200"/>
      </wsrm:RMAssertion>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

For a complete example of a `wsit-<package>.<service>.xml` file, see the `wsit-enabled-fromjava` example. You can use the `wsit-<package>.<service>.xml` file provided in the example as a reference for creating your own `wsit-<package>.<service>.xml` file.

Creating a Web Service From WSDL

Typically, you start from WSDL to build your web service if you want to implement a web service that is already defined either by a standard or an existing instance of the service. In either case, the WSDL already exists. The JAX-WS `wsimport` tool processes the existing WSDL document, either from a local copy on disk or by retrieving it from a network address or URL. For an example of using a web browser to access a service's WSDL, see Verifying Deployment (page 134).

When developing a web service starting from an existing WSDL, the process is actually simpler than starting from Java. This is because the policy assertions needed to enable various WSIT technologies are already embedded in the WSDL file. An example WSDL file is included in the `fromwsdl` sample provided with this tutorial at:

```
<INSTALL>/wsit-enabled-fromwsdl/etc/AddNumbers.wsdl
```

To Create a web service from WSDL, create the following source files:

- WSDL File (page 130)
- Web Service Implementation File (page 131)
- `custom-server.xml`
- `web.xml`
- `sun-jaxws.xml`
- `build.xml`
- `build.properties`

The following files are standard files required for JAX-WS. Examples of these files are provided in the `fromwsdl` sample directory.

- `custom-server.xml`
- `sun-jaxws.xml`
- `web.xml`

The `build.xml` and `build.properties` files are standard in any Ant build environment. Examples of these files are provided in the respective samples directories.

The sample files provided in this tutorial define a web service that takes two integers, adds them, and returns the result. If one of the integers is negative, an exception is returned.

WSDL File

You can create a WSDL file by hand or retrieve it from an existing web service by simply pointing a web browser at the web service's URL. The snippet shown below illustrates how to enable the WSIT Reliable Messaging technology in a WSDL file.

```
<wsp:Policy wsu:Id="AddNumbers_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsrm:RMAssertion>
        <wsrm:InactivityTimeout Milliseconds="600000"/>
        <wsrm:AcknowledgementInterval Milliseconds="200"/>
      </wsrm:RMAssertion>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

For a complete example of a WSDL file, see the `AddNumbers.wsdl` in the `from-wsdl` example. Another benefit of `AddNumbers.wsdl` file is that it shows how a WSIT-enabled WSDL is constructed. Therefore, you can use it as a reference when you create a WSDL file or modify an existing one.

Web Service Implementation File

The following file (`AddNumbersImpl.java`) shows how to implement a web service interface.

```
package fromwsdl.server;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService (endpointInterface=
    "fromwsdl.server.AddNumbersPortType")
public class AddNumbersImpl{
  @WebMethod(action="addNumbers")
  public int addNumbers (int number1, int number2)
       throws AddNumbersFault_Exception {
     if (number1 < 0 || number2 < 0) {
       String message = "Negative number cannot be added!";
       String detail = "Numbers: " + number1 + ", " + number2;
       AddNumbersFault fault = new AddNumbersFault ();
       fault.setMessage (message);
       fault.setFaultInfo (detail);
       throw new AddNumbersFault_Exception(message, fault);
     }
     return number1 + number2;
  }

  public void oneWayInt(int number) {
     System.out.println("Service received: " + number);
  }
}
```

Building and Deploying the Web Service

Once configured, you can build and deploy a WSIT-enabled web service in the same manner as you would build and deploy a standard JAX-WS web service.

The following topics describe how to perform this task:

- Building and Deploying a Web Service Created From Java (page 132)
- Building and Deploying a Web Service Created From WSDL (page 133)
- Deploying the Web Service to a Web Container (page 133)
- Verifying Deployment (page 134)

Building and Deploying a Web Service Created From Java

To build and deploy the web service, open a terminal window, go to the `<INSTALL>/wsit-enabled-fromjava/` directory and type the following:

```
ant server
```

This command calls the `server` target in `build.xml`, which builds and packages the application into a WAR file, `wsit-enabled-fromjava.war`, and places it in the `wsit-enabled-fromjava/build/war` directory. The `ant server` command also deploys the WAR file to the web container.

The `ant` command calls multiple tools to build and deploy the web service. The JAX-WS annotation processing tool (`apt`) processes the annotated source code and invokes the compiler itself, resulting in the class files for each of the Java source files. In the `wsit-enabled-fromjava` example, the `ant` target `build-server-java` in `build.xml` handles this portion of the process. Next, the individual class files are bundled together along with the web service's supporting configuration files into the application's WAR file. It is this file that is deployed to the web container by the `deploy` target.

During execution of the `server` target, you will see a warning message. The message refers to "Annotation types without processors". The warning is expected and does not indicate an abnormal situation. The text is included here for reference:

```
build-server-java:
  [apt] warning: Annotation types without processors:
    [javax.xml.bind.annotation.XmlRootElement,
     javax.xml.bind.annotation.XmlAccessorType,
     javax.xml.bind.annotation.XmlType,
     javax.xml.bind.annotation.XmlElement]
  [apt] 1 warning
```

Building and Deploying a Web Service Created From WSDL

To build and deploy the web service, open a terminal window, go to the *<INSTALL>*/wsit-enabled-fromjava/ directory, and type the following:

```
ant server
```

This command calls `wsimport`, which takes the WSDL description and generates a corresponding Java interface and other supporting classes. Then the Java compiler is called to compile both the user's code and the generated code. Finally, the class files are bundled together into the WAR file. To see the details of how this is done, see the `build-server-wsdl` and `create-war` targets in the `wsit-enabled-fromwsdl/build.xml` file.

Deploying the Web Service to a Web Container

As a convenience, invoking the `ant server` command builds the web service's WAR file and immediately deploys it to the web container. However, in some situations, such as after undeploying a web service, it may be useful to deploy the web service without rebuilding it.

For both scenarios, `wsit-enabled-fromjava` and `fromwsdl`, the resulting application is deployed in the same manner.

The following sections describe how to deploy on the different web containers:

- Deploying to GlassFish (page 133)
- Deploying to Apache Tomcat (page 134)

Deploying to GlassFish

For development purposes, the easiest way to deploy is to use the `autodeploy` facility of the GlassFish application server. To do so, you simply copy your application's WAR file to the `/autodeploy` directory for the domain to which you want to deploy. If you are using the default domain, `domain1`, which is set up by the GlassFish installation process, the appropriate directory path would be *<AS_HOME>*`/domains/domain1/autodeploy`.

The `build.xml` file which accompanies this example has a `deploy` target for GlassFish. To invoke that target, run the following command in the top-level directory of the respective examples, either `wsit-enabled-fromjava` or `wsit-enabled-fromwsdl`, as follows.

```
ant deploy
```

Deploying to Apache Tomcat

Apache Tomcat also has an `autoDeploy` feature that is enabled by Tomcat's out-of-the-box configuration settings. If you are not sure whether the `autoDeploy` is enabled, check `<`*`TOMCAT_HOME`*`>/conf/server.xml` for the value of `autoDeploy`. Assuming `autoDeploy` is enabled, you simply copy your application's WAR file to the `<`*`TOMCAT_HOME`*`>/webapps` directory

The `build.xml` file which accompanies this example has a `deploy` target for Tomcat. To invoke that target, run the following command in the top-level directory of the respective examples, either `wsit-enabled-fromjava` or `wsit-enabled-fromwsdl`, as follows. You need to use the `-Duse.tomcat=true` switch to make sure that the application is deployed to Tomcat, and not to the default server, which is GlassFish.

```
ant -Duse.tomcat=true deploy
```

Verifying Deployment

A basic test to verify that the application has deployed properly is to use a web browser to retrieve the application's WSDL from its hosting web container. The following URLs retrieve the WSDL from each of the two example services. If you are running your web browser and web container on different machines, you need to replace `localhost` with the name of the machine hosting your web service.

Note: Before testing, make sure your web container is running.

- `http://localhost:8080/wsit-enabled-fromjava/addnumbers?wsdl`
- `http://localhost:8080/wsit-enabled-fromwsdl/addnumbers?wsdl`

If the browser displays a page of XML tags, the web service has been deployed and is working. If not, check the web container log for any error messages related to the sample WAR you have just deployed. For GlassFish, the log can be found at *<AS_HOME>*`/domains/domain1/logs/server.log`. For Apache Tomcat, the appropriate log file can be found at *<TOMCAT_HOME>*`/logs/catalina.out`.

Creating a Web Service Client

Unlike developing a web service provider, creating a web service client application always starts with an existing WSDL file. This process is similar to the process you use to build a service from an existing WSDL file. The WSDL file that the client consumes already contains the WS–* policy assertions (and, in some cases, any value-added WSIT policy assertions that augment Sun's implementation, but can safely be ignored by other implementations). Most of the policy assertions are defined in the WS-* specifications. Sun's implementation processes these standard policy assertions.

The policy assertions describe any requirements from the server as well as any optional features the client may use. The WSIT build tools and run-time environment detect the WSDL's policy assertions and configure themselves appropriately, if possible. If an unsupported assertion is found, an error message describing the problem will be displayed.

Typically, you retrieve the WSDL directly from a web service provider using the `wsimport` tool. The `wsimport` tool then generates the corresponding Java source code for the interface described by the WSDL. The Java compiler, `javac`, is then called to compile the source into class files. The programming code uses the generated classes to access the web service.

The following sections describe how to create a web service client:

- Creating a Client from Java (page 136)
- Creating a Client from WSDL (page 138)

Creating a Client from Java

To create a client from Java, you must create the following files:

- Client Java File (fromjava) (page 136)
- Client Configuration File (fromjava) (page 137)
- `build.xml`
- `build.properties`

The `build.xml` and `build.properties` files are standard in any Ant build environment. Examples of these files are provided in the `wsit-enabled-fromjava` sample directory.

Client Java File (fromjava)

The client Java file defines the functionality of the web service client. The following code shows the `AddNumbersClient.java` file that is provided in the sample.

```
package fromjava.client;

import com.sun.xml.ws.Closeable;
import java.rmi.RemoteException;

public class AddNumbersClient {
  public static void main (String[] args) {
     AddNumbersImpl port = null;
     try {
        port = new
AddNumbersImplService().getAddNumbersImplPort();
        int number1 = 10;
        int number2 = 20;
        System.out.printf ("Invoking addNumbers(%d, %d)\n",
             number1, number2);
        int result = port.addNumbers (number1, number2);
        System.out.printf (
             "The result of adding %d and %d is %d.\n\n",
             number1, number2, result);

        number1 = -10;
        System.out.printf ("Invoking addNumbers(%d, %d)\n",
             number1, number2);
        result = port.addNumbers (number1, number2);
        System.out.printf (
             "The result of adding %d and %d is %d.\n",
                number1, number2, result);
        } catch (AddNumbersException_Exception ex) {
            System.out.printf (
                "Caught AddNumbersException_Exception: %s\n",
                ex.getFaultInfo ().getDetail ());
        } finally {
            ((Closeable)port).close();
        }
    }
}
```

This file specifies two positive integers that are to be added by the web service, passes the integers to the web service and gets the results from the web service via the `port.addNumbers` method, and prints the results to the screen. It then specifies a negative number to be added, gets the results (which should be an exception), and prints the results (the exception) to the screen.

Client Configuration File (fromjava)

The client configuration file defines the URL of the web service WSDL file. It is used by the web container `wsimport` tool to access and consume the WSDL and to build the stubs that are used to communicate with the web service.

The `custom-client.xml` file provided in the `wsit-enabled-fromjava` sample is shown below. The `wsdlLocation` and the `package name` xml tags are unique to each client and are highlighted in bold text

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bindings
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  wsdlLocation="http://localhost:8080/wsit-enabled-fromjava/
       addnumbers?wsdl"
  xmlns="http://java.sun.com/xml/ns/jaxws">
  <bindings node="wsdl:definitions">
    <package name="fromjava.client"/>
  </bindings>
</bindings>
```

Creating a Client from WSDL

To create a client from WSDL, you must create the following files:

- Client Java File (fromwsdl) (page 138)
- Client Configuration File (fromwsdl) (page 138)
- `build.xml`
- `build.properties`

The `build.xml` and `build.properties` files are standard in any Ant build environment. Examples of these files are provided in the `fromwsdl` sample directory.

Client Java File (fromwsdl)

The client Java file defines the functionality of the web service client. The same client java file is used with both samples, `wsit-enabled-fromjava` and `wsit-enabled-fromwsdl`. For more information on this file, see Client Java File (fromjava) (page 136).

Client Configuration File (fromwsdl)

This is a sample `custom-client.xml` file. The `wsdlLocation`, `package name`, and `jaxb:package name` xml tags are unique to each client and are highlighted in bold text

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bindings
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  wsdlLocation="http://localhost:8080/wsit-enabled-fromwsdl/
       addnumbers?wsdl"
  xmlns="http://java.sun.com/xml/ns/jaxws">
  <bindings node="ns1:definitions"
       xmlns:ns1="http://schemas.xmlsoap.org/wsdl/">
    <package name="fromwsdl.client"/>
  </bindings>
  <bindings node="ns1:definitions/ns1:types/xsd:schema
       [@targetNamespace='http://duke.org']"
       xmlns:xs="http://www.w3.org/2001/XMLSchema"
       xmlns:ns1="http://schemas.xmlsoap.org/wsdl/">
    <jaxb:schemaBindings>
```

```
          <jaxb:package name="fromwsdl.client"/>
        </jaxb:schemaBindings>
      </bindings>
    </bindings>
```

Building and Deploying a Client

To build and deploy a client for either of the examples provided in this tutorial, enter one of the following Ant commands in the top-level directory of the respective example, (either `wsit-enabled-fromjava` or `wsit-enabled-fromwsdl`) depending on which web container you are using:

For GlassFish: `ant client`

For the Apache Tomcat: `ant -Duse.tomcat=true client`

This command runs `wsimport`, which retrieves the web service's WSDL, and then it runs `javac` to compile the source.

Running a Web Service Client

To run a client for either of the examples provided in this tutorial, enter one of the following Ant commands in the top-level directory of the respective example, (either `wsit-enabled-fromjava` or `wsit-enabled-fromwsdl`) depending on which web container you are using:

For GlassFish: `ant run`

For the Apache Tomcat: `ant -Duse.tomcat=true run`

This command executes the `run` target, which simply runs Java with the name of the client's class, for example, `fromwsdl.client.AddNumbersClient`.

Undeploying a Web Service

During the development process, it is often useful to undeploy a web service. Undeploying a web service means to disable and remove it from the web container. Once the web service is removed, clients are no longer able to use the web service. Further, the web service will not restart without explicit redeployment by the user.

To undeploy from GlassFish, enter the following commands:

- `asadmin undeploy --user admin wsit-enabled-fromjava`
- `asadmin undeploy --user admin wsit-enabled-fromwsdl`

To undeploy from Apache Tomcat, enter the following commands:

- `rm $CATALINA_HOME/webapps/wsit-enabled-fromjava.war`
- `rm $CATALINA_HOME/webapps/wsit-enabled-fromwsdl.war`

8

Accessing WSIT Services Using WCF Clients

THIS chapter describes how to build and run a Microsoft Windows Communication Foundation (WCF) client that accesses the `addnumbers` service described in Chapter 7.

Creating a WCF Client

The process of creating a WCF C# client to the addnumbers service is similar to that for a Java programming language client. To create a WCF client you will:

1. Use the `svcutil.exe` tool to generate the C# proxy class and contracts for accessing the web service.
2. Create a client program that uses the generated files to make calls to the `addnumbers` web service.

Prerequisites to Creating the WCF Client

You must have the following software installed to create the WCF client:

- Microsoft Windows Software Development Kit (SDK) for July Community Technology Preview
- Microsoft .NET Framework 3.0 RTM
- the `csclient-enabled-fromjava.zip` example bundle, which you can download from `https://wsit.dev.java.net/source/browse/*checkout*/wsit/wsit/docs/howto/csclient-enabled-fromjava.zip`

You must also deploy the `addnumbers` service described in Chapter 7. You can download the service from `https://wsit.dev.java.net/source/browse/*checkout*/wsit/wsit/docs/howto/wsit-enabled-fromjava.zip`.

The Client Class

The client class uses a generated proxy class, `AddNumbersImpl`, to access the web service. The `port` instance variable stores a reference to the proxy class.

```
...
port = new AddNumbersImplClient("AddNumbersImplPort");
...
```

Then the web service operation `addNumbers` is called on `port`:

```
...
int result = port.addNumbers (number1, number2);
...
```

The following is the full `Client.cs` class:

```
using System;

class Client {
  static void Main(String[] args) {
    AddNumbersImplClient port = null;
    try {
      port = new AddNumbersImplClient("AddNumbersImplPort");
      int number1 = 10;
      int number2 = 20;

      Console.Write("Adding {0} and {1}. ", number1, number2);
      int result = port.addNumbers (number1, number2);
```

```
            Console.WriteLine("Result is {0}.\n\n",result);

            number1 = -10;
            Console.Write("Adding {0} and {1}. ", number1, number2);
            result = port.addNumbers (number1, number2);
            Console.WriteLine("Result is {0}.\n\n",result);
            port.Close();
         } catch (System.ServiceModel.FaultException e) {
            Console.WriteLine("Exception: " + e.Message);
            if (port != null) port.Close();
         }
      }
   }
```

Building and Running the Client

The example bundle contains all the files you need to build and run a WCF client that accesses a WSIT web service written in the Java programming language.

The `csclient-enabled-fromjava.zip` bundle contains the following files:

- `Client.cs`—the C# client class
- `build.bat`—the build batch file

Generating the Proxy Class and Configuration File

When creating a Java programming language client, you use the `wsimport` tool to generate the proxy and helper classes used by the client class to access the web service. When creating a WCF client, the `svcutil.exe` tool provides the same functionality as the `wsimport` tool. `svcutil.exe` generates the C# proxy class and contracts for accessing the service from a C# client program.

The example bundle contains a batch file, `build.bat`, that calls `svcutil.exe` to generate the proxy class. The command is:

```
svcutil /config:Client.exe.config http://localhost:8080/wsit-
enabled-fromjava/addnumbers?wsdl
```

Building the AddNumbers Client

The example bundle's `build.bat` file first generates the proxy class and configuration file for the client, then compiles the proxy class, configuration file, and `Client.cs` client class into the `Client.exe` executable file.

To run build.bat, do the following:

1. At a command prompt navigate the location where you extracted the example bundle.
2. If necessary, customize the `build.bat` file as described in Customizing the build.bat File below.
3. Enter the following command:

   ```
   build.bat
   ```

Customizing the build.bat File

To customize the build.bat file for your environment, do the following:

1. Open `build.bat` in a text editor.
2. On the first line enter the full path to the `svcutil.exe` tool. By default, it is installed at `C:\Program Files\Microsoft SDKs\Windows\v6.0\Bin`.
3. On the first line change the WSDL location URL if you did not deploy the `addnumbers` service to the local machine, or if the service was deployed to a different port than the default 8080 port number. For example, the following line sets the host name to `testmachine.example.com` and the port number to 8081:

   ```
      svcutil /config:Client.exe.config
   http://testmachine.example.com:8081/wsit-enabled-fromjava/
   addnumbers?wsdl
   ```

4. On line 2, change the location of the `csc.exe` C# compiler and the `System.ServiceModel` and `System.Runtime.Serialization` support DLLs if you installed the .NET 2.0 and 3.0 frameworks to non-default locations.

Running the AddNumbers Client

After the client has been built, run the client by following these steps:

1. At a command prompt navigate the location where you extracted the example bundle.
2. Enter the following command:

```
Client.exe
```

You will see the following output:

```
Adding 10 and 20. Result is 30.
Adding -10 and 20. Exception: Negative numbers can't
be added!
```

9 Data Contracts

THIS chapter describes guidelines for:

- Designing a XML schema exposed by a web service starting from Java
- Consuming a WCF service generated WSDL/XML schema when designing a Java client or Java web service
- Developing a Microsoft WCF client

A WSIT client/service uses JAXB 2.0 for XML serialization, generating XML schemas from Java classes and generating Java classes from XML schemas. A WCF client/service uses either `XmlSerializer` or `DataContractSerializer` for like tasks. JAXB 2.0 and the WCF XML serialization mechanisms differ in two fundamental ways. First, JAXB 2.0 supports all of XML schema. .NET's `DataContractSerializer` and `XmlSerializer` support different XML schema sets. Second, WCF's `XMLSerializer`/`DataContractSerializer` and JAXB 2.0 differ in their mapping of programming language datatypes to XML Schema constructs. As a result, a XML schema generated from a programming language on one platform and consumed on another platform may result in less than developer-friendly bindings. This chapter discusses some of the common databinding differences between the two systems and recommends ways to address them.

Web Service - Start from Java

This section provides guidelines for designing a XML schema exported by a Java web service designed starting from Java. JAXB 2.0 provides a rich set of annotations and types for mapping Java classes to different XML Schema constructs. The guidelines provide guidance on using JAXB 2.0 annotations and types so that developer friendly bindings may be generated by XML serialization mechanisms (svcutil) on WCF client.

Not all JAXB 2.0 annotations are included here; not all are relevant from an interoperability standpoint. For example, the annotation `@XmlAccessorType` provides control over default serialization of fields and properties in a Java class but otherwise has no effect on the on-the-wire XML representation or the XML schema generated from a Java class. Select JAXB 2.0 annotations are therefore not included here in the guidance.

The guidance includes several examples, which use the following conventions:

- prefix `xs:` is used to represent XML Schema namespace
- JAXB 2.0 annotations are defined in `javax.xml.bind.annotation` package but, for brevity, the package name has been omitted

DataTypes

Primitives and Wrappers

Guideline: Java primitive and wrapper classes map to slightly different XML schema representations. Therefore, .NET bindings will vary accordingly.

Example: A Java primitive type and its corresponding wrapper class

```
//-- Java code fragment
public class StockItem{
     public Double wholeSalePrice;
     public double retailPrice;
}

//--Schema fragment
<xs:complexType name="stockItem">
    <xs:sequence>
        <xs:element name="wholeSalePrice" type="xs:double"
```

```
            minOccurs="0"/>
             <xs:element name="retailPrice" type="xs:double"/>
                 </xs:sequence>
             </xs:complexType>
```

//-- .NET C# auto generated code from schema

```
             public partial class stockItem
             {
                  private double wholeSalePrice;
                  private bool wholeSalePriceFieldSpecified;
                  private double retailPrice;

                 public double wholeSalePrice
                 {
                      get{ return this.wholeSalePrice;}
                      set{this.wholeSalePrice=value}
                 }

                 public bool wholeSalePriceSpecified
                 {
                      get{ return
this.wholeSalePriceFieldSpecified;}
                   set{this.wholeSalePriceFieldSpecified=value}
                 }

                 public double retailPrice
                 {
                      get{ return this.retailPrice;}
                      set{this.retailPrice=value}
                 }
             }
```

//-- C# code fragment

```
                 stockItem s = new stockItem();
                 s.wholeSalePrice = Double.parse("198.92");
                 s.wholeSalePriceSpecified = true
                 s.retailPrice = Double.parse("300.25");
```

BigDecimal

Guideline: Limit decimal values to the range and precision of .NET's `System.decimal`.

`java.math.BigDecimal` maps to `xs:decimal`. .NET maps `xs:decimal` to `System.decimal`. These two data types support different range and precision. `java.math.BigDecimal` supports arbitrary precision. `System.decimal` does not. For interoperability use only values within the range and precision of `System.decimal`. (See `System.decimal.Minvalue` and `System.decimal.Maxvalue`.) Any values outside of this range require a customized .NET client.

Example: `BigDecimal` **usage**

//--- Java code fragment

```
  public class RetBigDecimal {
      private BigDecimal arg0;

      public BigDecimal getArg0() { return this.arg0; }
      public void setArg0(BigDecimal arg0) { this.arg0 = arg0; }
   }
```

//--- Schema fragment

```
  <xs:complexType name="retBigDecimal">
      <xs:sequence>
         <xs:element name="arg0" type="xs:decimal" minOccurs="0"/>
      </xs:sequence>
  </xs:complexType>
```

//--- .NET auto generated code from schema

```
  public partial class retBigDecimal{
      private decimal arg0Field;
      private bool arg0FieldSpecified;

      public decimal arg0 {
         get { return this.arg0Field;}
         set { this.arg0Field = value;}
      }

      public bool arg0Specified {
         get { return this.arg0FieldSpecified;}
         set { this.arg0FieldSpecified = value;}
      }
  }
```

//--- C# code fragment

```
      System.CultureInfo engCulture = new System.CultureInfo("en-
US");
      retBigDecimal bd = new retBigDecimal();
      bd.arg0 = System.decimal.MinValue;
```

```
     retBigDecimal negBd = new retBigDecimal();
    negBd = System.decimal.Parse("-0.0", engCulture);
```

java.net.URI

Guideline: Use the `@XmlSchemaType` annotation for a strongly typed binding to a .NET client generated with the `DataContractSerializer`.

`java.net.URI` maps to `xs:string`. .NET maps `xs:string` to `System.string`. Annotation `@XmlSchemaType` can be used to define a more strongly typed binding to a .NET client generated with the `DataContractSerializer`. `@XmlSchemaType` can be used to map `java.net.URI` to `xs:anyURI`. .NET's `DataContractSerializer` and `XmlSerializer` bind `xs:anyURI` differently:

- `DataContractSerializer` binds `xs:anyURI` to .NET type `System.Uri`
- `XmlSerializer` binds `xs:anyURI` to .NET type `System.string`

Thus, the above technique only works if the WSDL is processed using `DataContractSerializer`.

Example: `@XmlSchemaType` and `DataContractSerializer`

```
// Java code fragment
public class PurchaseOrder
{
  @XmlSchemaType(name=”anyURI”)
  public java.net.URI uri;
}

//-- Schema fragment
<xs:complexType name=”purchaseOrder”>
  <xs:sequence>
     <xs:element name=”uri” type=”xs:anyURI” minOccurs=”0”/>
  </xs:sequence>
</xs:complexType>

//--- .NET auto generated code from schema
//--- Using svcutil.exe /serializer:DataContractSerializer <wsdl file>
  public partial class purchaseOrder : object,
          System.Runtime.Serialization.IExtensibleDataObject
        {

                  private System.Uri uriField;

                  //-- ..... other gernerated code ........
public System.Uri uri
```

```
            {
                get { return this.uriField; }
                set { this.uriField = value; }
            }
        }
```

//--- C# code fragment

```
   purchaseOrder tmpU = new purchaseOrder()
   tmpU.uri = new System.Uri("../Hello",
System.UriKind.Relative);
```

Example: `@XmlSchemaType` **and** `XmlSerializer`

// Java code fragment

```
public class PurchaseOrder
{
  @XmlSchemaType(name=”anyURI”)
  public java.net.URI uri;
}
```

//--- .NET auto generated code from schema
//--- Using svcutil.exe /serializer:XmlSerializer <wsdl file>

```
   public partial class purchaseOrder
    {
        private string uriField;
        public string uri
        {
            get { return this.uriField; }
            set { this.uriField = value; }
        }
    }
```

//--- C# code fragment

```
        purchaseOrder tmpU = new purchaseOrder()
        tmpU.uri = "mailto:mailto:mduerst@ifi.unizh.ch";
```

Duration

Guideline: Use .NET’s `System.Xml.XmlConvert` to generate a lexical representation of `xs:duration` when the binding is to a type of `System.string`.

`javax.xml.datatype.Duration` maps to `xs:duration`. .NET maps `xs:duration` to a different datatype for `DataContractSerializer` and `XmlSerializer`.

- `DataContractSerializer` binds `xs:duration` to .NET `System.TimeSpan`.

- `XmlSerializer` binds `xs:duration` to .NET `System.string`.

When `xs:duration` is bound to .NET `System.string`, the `string` value must be a lexical representation for `xs:duration`. .NET provides utility `System.Xml.XmlConvert` for this purpose.

Example: Mapping `xs:duration` **using** `DataContactSerializer`

```
//-- Java code fragment
  public class PurchaseReport {
     public javax.xml.datatype.Duration period;
  }

  //-- Schema fragment
  <xs:complexType name="purchaseReport">
    <xs:sequence>
      <xs:element name="period" type="xs:duration" 
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  //-- .NET auto generated code from schema
 //-- Using svcutil.exe /serializer:DataContractSerializer <wsdl file>
    public partial class purchaseReport: object, 
        System.Runtime.Serialization.IExtensibleDataObject
    {
        private System.TimeSpan periodField;
        //-- ..... other gernerated code ........
        public System.TimeSpan period
        {
            get { return this.periodField; }
            set { this.periodField = value; }
        }
    }

  //-- C# code fragment
  purchaseReport tmpR = new purchaseReport();
  tmpR.period = new System.TimeSpan.MaxValue;
```

Example: Mapping `xs:duration` **using** `XmlSerializer`

```
//-- .NET auto generated code from schema
//-- Using svcutil.exe /serializer:XmlSerializer <wsdl file>
 public partial class purchaseReport
 {
       private string periodField;
       public string period
```

```
        {
            get { return this.periodField; }
            set { this.periodField = value; }
        }
 }
```

//-- C# code fragment

```
purchaseReport tmpR = new purchaseReport();
tmpR.period = System.Xml.XmlConvert.ToString(new
System.TimeSpan(23, 0,0));
```

Binary Types

`java.awt.Image`, `javax.xml.transform.Source`, and `javax.activation.DataHandler` map to `xs:base64Binary`. .NET maps `xs:base64Binary` to `byte[]`.

JAXB 2.0 provides the annotation `@XmlMimeType`, which supports specifying the content type, but .NET ignores this information.

Example: Mapping `java.awt.Image` **without** `@XmlMimeType`

//-- Java code fragment

```
  public class Claim{
        public java.awt.Image photo;
  }
```

//-- Schema fragment

```
  <xs:complexType name="claim">
    <xs:sequence>
      <xs:element name="photo" type="xs:base64Binary"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

//-- .NET auto generated code from schema

```
  public partial class claim : object,
        System.Runtime.Serialization.IExtensibleDataObject
  {
        private byte[] photoField;
       //-- ..... other gernerated code .......
        public byte[] photo
        {
            get { return this.photoField; }
            set { this.photoField = value; }
        }
    }
```

//-- C# code fragment

```
    try
    {
       claim tmpC = new claim();

       System.IO.FileStream f = new System.IO.FileStream(
          "C:\\icons\\circleIcon.gif", System.IO.FileMode.Open);
       int cnt = (int)f.Length;
       tmpC.photo = new byte[cnt];
       int rCnt = f.Read(tmpC.photo, 0, cnt);

    }
    catch (Exception e)
    {
       Console.WriteLine(e.ToString());
    }
```

Example: Mapping `java.awt.Image` **with** `@XmlMimeType`

//-- Java code fragment

```
public class Claim{
       @XmlMimeType("image/gif")
        public java.awt.Image photo;
}
```

//-- Schema fragment

```
  <xs:complexType name="claim">
    <xs:sequence>
     <xs:element name="photo" ns1:expectedContentTypes="image/
gif"
       type="xs:base64Binary" minOccurs="0"
       xmlns:ns1="http://www.w3.org/2005/05/xmlmime"/>
    </xs:sequence>
  </xs:complexType>
```

//-- Using the @XmlMimeType annotation doesn’t change .NET
//--auto generated code

```
    public partial class claim : object,
        System.Runtime.Serialization.IExtensibleDataObject
    {
        private byte[] photoField;
       //-- ..... other gernerated code .......
        public byte[] photo
```

```
            {
                get { return this.photoField; }
                set { this.photoField = value; }
            }
        }
```

//-- This code is unchanged by the different schema
//-- C# code fragment

```
    try
    {
       claim tmpC = new claim();

       System.IO.FileStream f = new System.IO.FileStream(
          "C:\\icons\\circleIcon.gif", System.IO.FileMode.Open);
       int cnt = (int)f.Length;
       tmpC.photo = new byte[cnt];
       int rCnt = f.Read(tmpC.photo, 0, cnt);

    }
    catch (Exception e)
    {
       Console.WriteLine(e.ToString());
    }
```

XMLGregorianCalendar

Guideline: Use `java.xml.datatype.XMLGregorianCalendar` instead of `java.util.Date` and `java.util.Calendar`.

`XMLGregorianCalendar` supports the following XML schema calendar types: `xs:date`, `xs:time`, `xs:dateTime`, `xs:gYearMonth`, `xs:gMonthDay`, `xs:gYear`, `xs:gMonth`, and `xs:gDay`. It is statically mapped to `xs:anySimpleType`, the common schema type from which all the XML schema calendar types are derived. .NET maps `xs:anySimpleType` to `System.string`.

`java.util.Date` and `java.util.Calendar` map to `xs:dateTime`, but don't provide as complete XML support as `XMLGregorianCalendar` does.

Guideline: Use annotation `@XmlSchemaType` for a strongly typed binding of `XMLGregorianCalendar` to one of the XML schema calendar types.

Example: `XmlGregorianCalendar` **without** `@XmlSchemaType`

//-- Java code fragment
```
 public class PurchaseOrder{
     public javax.xml.datatype.XMLGregorianCalendar orderDate;
  }
```

//-- Schema fragment
```
  <xs:complexType name="purchaseOrder">
    <xs:sequence>
      <xs:element name="orderDate" type="xs:anySimpleType"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

//-- .NET auto generated code from schema
```
  public partial class purchaseOrder
  {
      private string orderDateField;
      public string orderDate
      {
          get { return this.orderDateField; }
          set { this.orderDateField = value; }
      }
  }
```

//-- C# code fragment
```
  purchaseOrder tmpP = new purchaseOrder();
  tmpP.orderDate = System.Xml.XmlConvert.ToString(
  System.DateTime.Now,
System.Xml.XmlDateTimeSerializerMode.RoundtripKind);
```

Example: `XMLGregorianCalendar` **with** `@XmlSchemaType`

//-- Java code fragment
```
public class PurchaseOrder{
     @XmlSchemaType(name="dateTime")
     public javax.xml.datatype.XMLGregorianCalendar orderDate;
 }
```

//-- Schema fragment
```
 <xs:complexType name="purchaseOrder">
    <xs:sequence>
      <xs:element name="orderDate" type="xs:dateTime"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

//-- .NET auto generated code from schema
```
  public partial class purchaseOrder : object,
    System.Runtime.Serialization.IExtensibleDataObject
  {
     private System.Runtime.Serialization.ExtensionDataObject
extensionDataField;
      private System.DateTime orderDateField;

      public System.Runtime.Serialization.ExtensionDataObject
ExtensionData
      {
          get { return this.extensionDataField; }
          set { this.extensionDataField = value; }
      }

      public System.DateTime orderDate
      {
          get { return this.orderDateField; }
          set { this.orderDateField = value; }
      }
  }
```

//-- C# code fragment
```
  purchaseOrder tmpP = new purchaseOrder();
  tmpP.orderDate = System.DateTime.Now;
```

UUID

Guideline: Use Leach-Salz variant of UUID at runtime.

`java.util.UUID` maps to schema type `xs:string`. .NET maps `xs:string` to `System.string`. The constructors in `java.util.UUID` allow any variant of UUID to be created. Its methods are for manipulation of the Leach-Salz variant.

Example: Mapping UUID

//-- Java code fragment
```
public class ReportUid{
   public java.util.UUID uuid;
 }
```

//-- Schema fragment
```
  <xs:complexType name="reportUid">
    <xs:sequence>
      <xs:element name="uuid" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

//-- .NET auto generated code from schema
```
  public partial class reportUid: object,
      System.Runtime.Serialization.IExtensibleDataObject
  {
     private System.Runtime.Serialization.ExtensionDataObject
extensionDataField;
      private string uuidField;

      public System.Runtime.Serialization.ExtensionDataObject
ExtensionData
      {
          get { return this.extensionDataField; }
          set { this.extensionDataField = value; }
      }

      public string uuid
      {
          get { return this.uuidField; }
          set { this.uuidField = value; }
      }
  }
```

//-- C# code fragment
```
  reportUid tmpU = new reportUid();
  System.Guid guid = new System.Guid("06b7857a-05d8-4c14-b7fa-
822e2aa6053f");
  tmpU.uuid = guid.ToString();
```

Type Variable

A typed variable maps to `xs:anyType`. .NET maps `xs:anyType` to `System.Object`.

Example: Using a typed variable

```
// Java class
public class Shape <T>
{
  private T xshape;

  public Shape() {};
  public Shape(T f)
  {
    xshape = f;
  }
}

<xs:complexType name=”shape”>
  <xs:sequence>
    <xs:element name=”xshape” type=”xs:anyType”
minOccurs=”0”/>
  </xs:sequence>
</xs:complexType>

// C# code generated by svcutil
public partial class shape
{
  private object xshapeField;

  public object xshape
  {
    get { return this.xshapeField; }
    set { this.xshapeField = value; }
  }
}
```

Collections

Java collections types - `java.util.Collection` and its subtypes, array, List, and parameterized collection types (e.g. `List<Integer>`) can be mapped to XML schema in different ways and can be serialized in different ways. The following examples show .NET bindings.

List of nillable elements

By default, a collection type such as `List<Integer>` maps to a XML schema construct that is a repeating unbounded occurrence of an optional and nillable element. .NET binds the XML schema construct to `System.Nullable<int>[]`. The element is optional and nillable. However, when marshalling JAXB marshaller will always marshal a null value using `xsi:nil`.

Example: Collection to a list of nillable elements

//-- Java code fragment
```
@XmlRootElement(name="po")
public PurchaseOrder {
  public List<Integer> items;
}
```

//-- Schema fragment
```
<xs:element name="po" type="purchaseOrder">
<xs:complexType name="purchaseOrder">
  <xs:sequence>
     <xs:element name="items" type="xs:int" nillable="true"
                  minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

//--- JAXB XML serialization
```
<po>
  <items> 1 </items>
  <items> 2 </items>
  <items> 3 </items>
</po>

<po>
  <items> 1 </items>
  <items xsi:nil=true/>
  <items> 3 </items>
</po>
```

//-- .NET auto generated code from schema
```
partial class purchaseOrder {
  private System.Nullable<int>[] itemsField;

  public System.Nullable<int>[] items
  {
     get { return this.itemsField; }
     set { this.itemsField = value; }
  }
}
```

List of optional elements

This is the same as above except that a collection type such as `List<Integer>` maps to a repeating unbounded occurrence of an optional (`minOccurs="0"`) but not nillable element. This in turn binds to .NET type `int[]`. This is more developer friendly. However, when marshalling, JAXB will marshal a null value within the `List<Integer>` as a value that is absent from the XML instance.

Example: Collection to a list of optional elements

```
//-- Java code fragment
@XmlRootElement(name="po")
public PurchaseOrder {
  @XmlElement(nillable=false)
  public List<Integer> items;
}

//-- Schema fragment
<xs:element name="po" type="purchaseOrder">
<xs:complexType name="purchaseOrder">
  <xs:sequence>
    <xs:element name="items" type="xs:int"
                minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

// .NET auto generated code from schema
partial class purchaseOrder {
  private int[] itemsField;

  public int[] items
  {
    get { return this.itemsField; }
    set { this.itemsField = value; }
  }
}
```

List of values

A collection such as `List<Integer>` can be mapped to a list of XML values (i.e. a XML schema list simple type) using annotation `@XmlList`. .NET maps list simple type to a .NET `System.string`.

Example: Collection to a list of values using `@XmlList`

```
//-- Java code fragment
@XmlRootElement(name=”po”)
public PurchaseOrder {
  @XmlList public List<Integer> items;
}

//-- Schema fragment
<xs:element name=”po” type=”purchaseOrder”>
<xs:complexType name=”purchaseOrder”>
  <xs:element name=”items” minOccurs=”0”>
    <xs:simpleType>
      <xs:list itemType=”xs:int”/>
    </xs:simpleType>
  </xs:element>
</xs:complexType>

//-- XML serialization
<po>
  <items> 1 2 3 </items>
</po>

// .NET auto generated code from schema
partial class purchaseOrder {
  private string itemsField;

  public string items
  {
    get { return this.itemsField; }
    set { this.itemsField = value; }
  }
```

Arrays

Example: Single and multidimensional Arrays

```
//-- Java code fragment
public class FamilyTree {
    public Person[] persons;
    public Person[][] family;
}

// .NET auto generated code from schema
public partial class familyTree
{
    private person[] persons;
    private person[][] families;

    public person[] persons
    {
        get { return this.membersField; }
        set { this.membersField = value; }
    }

    public person[][] families {
        get { return this.familiesField; }
        set { this.familiesField = value; }
    }
}
```

Fields/Properties

The following guidelines apply to mapping of Javabean properties and Java fields, but for brevity Java fields are used.

@XmlElement

The `@XmlElement` annotation maps a property/field to an XML element. This is also the default mapping in the absence of any other JAXB 2.0 annotations. The annotation parameters in `@XmlElement` can be used to specify whether the element is optional or required, nillable or not. The following examples illustrate the corresponding bindings in the .NET client.

Example: Map a field/property to a nillable element

//-- Java code fragment
```
public class PurchaseOrder {

  // Map a field to a nillable XML element
  @javax.xml.bind.annotation.XmlElement(nillable=true)
  public java.math.BigDecimal price;

}
```

//-- Schema fragment
```
<xs:complexType name="purchaseOrder">
  <xs:sequence>
    <xs:element name="price" type="xs:decimal"
                nillable="true" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

// .NET auto generated code from schema
```
public partial class purchaseOrder {
  private System.Nullable<decimal> priceField;
  private bool priceFieldSpecified;

  public decimal price
  {
    get { return this.priceField; }
    set { this.priceField = value; }
  }

  public bool priceSpecified {
  {
    get { return this.priceFieldSpecified; }
    set { this.priceFieldSpecified = value;}
  }
```

Example: Map property/field to a nillable, required element

//-- Java code fragment
```
public class PurchaseOrder {

  // Map a field to a nillable XML element
  @XmlElement(nillable=true, required=true)
  public java.math.BigDecimal price;

}
```

//-- Schema fragment

```
<xs:complexType name="purchaseOrder">
  <xs:sequence>
    <xs:element name="price" type="xs:decimal"
                nillable="true" minOccurs="1" />
  </xs:sequence>
</xs:complexType>
```

// .NET auto generated code from schema

```
public partial class purchaseOrder {
  private System.Nullable<decimal> priceField;

  public decimal price
  {
    get { return this.priceField; }
    set { this.priceField = value; }
  }

  }
```

@XmlAttribute

A property/field can be mapped to an XML attribute using `@XmlAttribute` annotation. .NET binds an XML attribute to a property.

Example: Mapping field/property to XML attribute

//-- Java code fragment

```
public class UKAddress extends Address {
  @XmlAttribute
  public int exportCode;
}
```

//-- Schema fragment

```
<! XML Schema fragment -->
<xs:complexType name="ukAddress">
  <xs:complexContent>
    <xs:extension base="tns:address">
      <xs:sequence/>
      <xs:attribute name="exportCode" type="xs:int"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

// .NET auto generated code from schema

```
public partial class ukAddress : address
{
  private int exportCodeField;
  public int exportCode
  {
    get { return this.exportCodeField; }
    set { this.exportCodeField = value; }
  }
}
```

@XmlElementRefs

Guideline: `@XmlElementRefs` maps to a `xs:choice`. This binds to a property with name "item" in the C# class. If there is another field/property named "item" in the Java class, there will be a name clash that .NET will resolve by generating `name`. To avoid the nameclash, either change the name or use customization, for example `@XmlElement(name="foo")`.

Example: Mapping a field/property using @XmlElementRefs

//-- Java code fragment

```
public class PurchaseOrder {
  @XmlElementRefs({
    @XmlElementRef(name="plane", type=PlaneType.class),
    @XmlElementRef(name="auto", type=AutoType.class)})
  public TransportType shipBy;
}

@XmlRootElement(name="plane")
public class PlaneType extends TransportType {}

@XmlRootElement(name="auto")
```

```
public class AutoType extends TransportType { }

@XmlRootElement
public class TransportType { ... }
```

//-- Schema fragment

```
<!-- XML schema generated by wsgen -->
<xs:complexType name=”purchaseOrder”>
  <xs:choice>
    <xs:element ref=”plane”/>
    <xs:element ref=”auto”/>
  </xs:choice>
</xs:complexType>

<!-- XML global elements -->
<xs:element name=”plane” type=”autoType” />
<xs:element name=”auto” type=”planeType” />

<xs:complexType name=”autoType”>
  <!-- content omitted - details not relevant to example -->
</xs:complexType>

</xs:complexType name=”planeType”>
  <!-- content omitted - details not relevant to example -->
</xs:complexType>
```

// .NET auto generated code from schema

```
public partial class purchaseOrder {
   private transportType itemField;

  [System.Xml.Serialization.XmlElementAttribute("auto",
typeof(autoType), Order=4)]
  [System.Xml.Serialization.XmlElementAttribute("plane",
typeof(planeType), Order=4)]
  public transportType Item
  {
    get { return this.itemField; }
    set { this.itemField = value; }
  }

public partial class planeType { ... } ;
public partial class autoType { ... } ;
```

Class

A Java class can be mapped to different XML schema type and/or an XML element. The following guidelines apply to the usage of annotations at the class level.

@XmlType - Anonymous type

Guideline: Prefer mapping class to named XML schema type rather than an anonymous type for a better .NET type binding.

The `@XmlType` annotation is used to customize the mapping of a Java class to an anonymous type. .NET binds an anonymous type to a .NET class - one per reference to the anonymous type. Thus, each Java class mapped to an anonymous type can generate multiple classes on the .NET client.

Example: Mapping a Java class to an anonymous type using `@XmlType`

```
//-- Java code fragment
public class PurchaseOrder {
     public java.util.List<Item> item;
}

@XmlType(name="")
public class Item {
  public String productName;
  ...
}

//-- Schema fragment
<xs:complexType name=”purchaseOrder”>
  <xs:sequence>
     <xs:element name=”item”>
        <xs:complexType>
           <xs:sequence>
              <xs:element name=”productName” type=”xs:string”/>
           </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

// C# code generated by svcutil

```
public partial class purchaseOrder
{
     private purchaseOrderItem[] itemField;

     System.Xml.Serialization.XmlElementAttribute("item",
Form=System.Xml.Schema.XmlSchemaForm.Unqualified,
IsNullable=true, Order=0)]
     public purchaseOrderItem[] item
     {
        get {
           return this.itemField;
        }

        set {
           this.itemField = value;
        }
     }
```

// .NET auto generated code from schema

```
  public partial class purchaseOrderItem
  {
     private string productNameField;
        public string productName {
           get { return this.productNameField; }
           set { this.productNameField = value; }
        }
  }
```

@XmlType - xs:all

Guideline: Avoid using `XmlType(propOrder=:{})`.

`@XmlType(propOrder={})` maps a Java class to a XML Schema complex type with `xs:all` content model. Since XML Schema places severe restrictions on `xs:all`, the use of `@XmlType(propOrder={})` is therefore not recommended. So, the following example shows the mapping of a Java class to `xs:all`, but the corresponding .NET code generated by `svcutil` is omitted.

Example: Mapping a class to `xs:all` using `@XmlType`

//-- Java code fragment
```
@XmlType(propOrder={})
public class USAddress {
  public String name;
  public String street;
}
```

//-- Schema fragment
```
<xs:complexType name="USAddress">
  <xs:all>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="street" type="xs:string"/>
    ...
  </xs:all>
</xs:complexType>
```

@XmlType - simple content

Guideline: A class can be mapped to a `complexType` with a `simpleContent` using `@XmlValue` annotation. .NET binds the Java property annotated with `@XmlValue` to a property with name “value”.

Example: Class to `complexType` with `simpleContent`

//-- Java code fragment
```
public class InternationalPrice
{
  @XmlValue
  public java.math.BigDecimal price;

  @XmlAttribute public String currency;
}
```

//-- Schema fragment
```
<xs:complexType name=”internationalPrice”>
  <xs:simpleContent>
    <xs:extension base=xs:decimal”>
```

```
      xs:attribute name="currency" type="xs:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

// .NET auto generated code from schema

```
public partial class internationalPrice
{
  private string currencyField;
  private decimal valueField;
  public string currency
  {
    get { return this.currencyField; }
    set { this.currencyField = value;}
  }

  public decimal Value
  {
    get { return this.valueField; }
    set { this.valueField = value;}
  }
}
```

Open Content

JAXB 2.0 supports the following annotations for defining open content. (Open content allows content not statically defined in XML schema to occur in an XML instance):

- `@XmlAnyElement` - which maps to `xs:any`, which binds to .NET type `System.Xml.XmlElement[]`.
- `@XmlAnyAttribute` - which maps to `xs:anyAttribute`, which binds to .NET type `System.Xml.XmlAttribute[]`.

Example: Using `@XmlAnyElement` for open content

//-- Java code fragment

```
@XmlType(propOrder={"name", "age", "oc"})
public class OcPerson {
  @XmlElement(required=true)
  public String name;
  public int age;
  // Define open content
  @XmlAnyElement
  public List<Object> oc;
}
```

//-- Schema fragment

```
<xs:complexType name="ocPerson">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="age" type="xs:int"/>
    <xs:any minOccurs="0" maxOccurs="unbounded">
  </xs:sequence>
</xs:complexType>
```

// .NET auto generated code from schema

```
public class ocPerson
{
  private String name;
  private int age;
  private System.Xml.XmlElement[] anyField;<

  public String name { ... }
  public int age { ... }


  public System.Xml.XmlElement[] Any {
  {
    get { return this.anyField; }
    set { this.anyField = value; }
  }
}
```

Example: Open content using `@XmlAnyAttribute`

//-- Java code fragment

```
@XmlType(propOrder={“name”, “age”}
public class OcPerson {
  public String name;
  public int age;
  // Define open content
  @XmlAnyAttribute
  public java.util.Map oc;
}
```

//-- Schema fragment

```
<xs:complexType name=”ocPerson”>
  <xs:sequence>
    <xs:element name=”name” type=”xs:string”/>
    <xs:element name=”age” type=”xs:int”/>
  </xs:sequence>
  <xs:anyAttribute/>
</xs:complexType>
```

// .NET auto generated code from schema

```
public class ocPerson
{
  private String name;
  private double age;
  private System.Xml.XmlAttribute[] anyAttrField;<

  public String name { ... }
  public double age { ... }


  public System.Xml.XmlElement[] anyAttr {
  {
    get { return this.anyAttrField; }
    set { this.anyAttrField = value; }
  }
}
```

Enum Type

A Java `enum` type maps to an XML schema type constrained by enumeration facets. This, in turn, binds to .NET type `enum` type.

Example: Java `enum` -> `xs:simpleType` (with `enum` facets) -> .NET `enum`

//-- Java code fragment
```
public enum USState {MA, NH}
```

//-- Schema fragment
```
<xs:simpleType name=”usState”>
  <xs:restriction base=”xs:string”>
    <xs:enumeration value=”NH” />
    <xs:enumeration value=”MA” />
  </xs:restriction>
</xs:simpleType>
```

// .NET auto generated code from schema
```
public enum usState { NH, MA }
```

Package

The following package level JAXB annotations are relevant from an interoperability standpoint:

- `@XmlSchema` – customizes the mapping of package to XML namespace.
- `@XmlSchemaType` – customizes the mapping of XML schema built-in type. The `@XmlSchemaType` annotation can also be used at the property/field level, as was seen in the previous example XMLGregorianCalendar (page 156).

@XmlSchema

A package is mapped to an XML namespace. The following attributes of the XML namespace can be customized using the `@XmlSchema` annotation parameters:

- `elementFormDefault` using `@XmlSchema.elementFormDefault()`
- `attributeFormDefault` using `@XmlSchema.attributeFormDefault()`
- `targetNamespace` using `@XmlSchema.namespace()`
- Associate namespace prefixes with the XML namespaces using the `@XmlSchema.ns()` annotation

These XML namespace attributes are bound to .NET serialization attributes (for example, `XmlSerializer` attributes).

Not Recommended Annotations

Any JAXB 2.0 annotation can be used but the following are not recommended:

- The `javax.xml.bind.annotation.XmlElementDecl` annotation is used to provide complete XML schema support.
- The `@XmlID` and `@XmlIDREF` annotations are used for XML object graph serialization, which is not well supported.

Web Service - Start from WSDL

The following guidelines apply when designing a Java web service starting from a WSDL:

1. If the WSDL was generated by `DataContractSerializer`, enable JAXB 2.0 customizations described in Customizations for WCF Service WSDL (page 177)”. The rationale for the JAXB 2.0 customizations is described in the same section.
2. If the WSDL is a result of contract first approach, verify that the WSDL can be processed by either the `DataContractSerializer` or `XmlSerializer` mechanisms.

   The purpose of this step is to ensure that the WSDL uses only the set of XML schema features supported by JAXB 2.0 or .NET serialization mechanisms. JAXB 2.0 was designed to support all the XML schema features. The WCF serialization mechanisms, `DataContractSerializer` and `XmlSerializer`, provide different levels of support for XML schema features. Thus, the following step will ensure that the WSDL/schema file can be consumed by the WCF serialization mechanisms.

   ```
   svcutil <wsdl-file>
   ```

   `svcutil.exe` tool, by default, uses `DataContractSerializer` but falls back to `XmlSerializer` if it encounters an XML schema construct not supported by `XmlFormatter`.

Java Client

A Java client is always developed starting from a WSDL. See section, “Customizations for WCF Service WSDL (page 177)” for guidelines.

Customizations for WCF Service WSDL

When developing either a Java web service or a Java client from a WCF service WSDL generated using `DataContractSerializer`, the following JAXB 2.0 customizations are useful and/or required.

- `generateElementProperty`
- `mapSimpleTypeDef`

The following sections explain the use and rationale of these customizations.

generateElementProperty

WCF service WSDL generated from a programming language such as C# using `DataContractSerializer` may contain XML Schema constructs which result in `JAXBElement<T>` in generated code. A `JAXBElement<T>` type can also sometimes be generated when a WSDL contains advanced XML schema features such as substitution groups or elements that are both optional and nillable. In all such cases, `JAXBElement<T>` provides roundtripping support of values in XML instances. However, `JAXBElement<T>` is not natural to a Java developer. So the `generateElementProperty` customization can be used to generate an alternate developer friendly but lossy binding. The different bindings along with the trade-offs are discussed below.

Default Binding

The following is the default binding of an optional `(minOccurs=”0”)` and `nillable(nillable=”true”)` element:

```
<!-- XML schema fragment
<xs:element name=”person” type=”Person”
<xs:complexType name="Person">
  <xs:sequence>
    <xs:element name="name" type="xs:string"
        nillable="true" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

// Binding
public class Person {
  JAXBElement<String> getName() {...};
  public void setName(JAXBElement<String> value) {...}
}
```

Since the XML element "name" is both optional and nillable, it can be represented in an XML instance in one of following ways:

```
<!-- Absence of element name-->
<person>
  <-- element name is absent -->
</person>

<!-- Presence of an element name -->
<person>
  <name xsi:nil="true"/>
</person>
```

The `JAXBElement`<String> type roundtrips the XML representation of "name" element across an unmarshal/marshal operation.

Customized Binding

When `generateElementProperty` is `false`, the binding is changed as follows:

```
// set JAXB customization generateElementProperty="false"/>
public class Person {
  String getName() {...}
  public void setName(String value) {...}
}
```

The above binding is more natural to Java developer than `JAXBElement<String>`. However, it does not roundtrip the value of <`name`>.

JAXB 2.0 allows `generateElementProperty` to be set:

- Globally in <`jaxb:globalBindings`>
- Locally in <`jaxb:property`> customization

When processing a WCF service WSDL, it is recommended that the `generateElementProperty` customization be set in <`jaxb:globalBindings`>:

```
<jaxb:bindings version="2.0"
         xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
         xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <jaxb:bindings schemaLocation="schema-importedby-wcfsvcwsdl"
                 node="/xs:schema">
    <jaxb:globalBindings generateElementProperty="false"/>
  </jxb:bindings>
```

Note: The `generateElementProperty` attribute was introduced in JAXB 2.1.

mapSimpleTypeDef

XML Schema Part 2: Datatype defines facilities for defining datatypes for use in XML Schemas. .NET platform introduced the CLR types for some of the XML schema datatypes as described in Table 9–1.

Table 9–1 CLR to XML Schema Type Mapping

CLR Type	XML Schema Type
`byte`	`xs:unsignedByte`
`uint`	`xs:unsignedInt`
`ushort`	`xs:unsignedShor`
`ulong`	`xs:unsignedLong`

However, there are no corresponding Java types that map to the XML Schema types listed in Table 9–1. Furthermore, JAXB 2.0 maps these XML schema types to Java types that are natural to Java developer. However, this results in a mapping that is not one-to-one. For example:

- `xs:int` -> `int`
- `xs:unsignedShort` -> `int`

The lack of a one-to-one mapping means that when XML Schema types shown in Table 9–1 are used in an `xsi:type` construct, they won't be preserved by default across an unmarshal followed by marshal operation. For example:

```
// C# web method
public Object retObject(Object objvalue);

// Java web method generated from WCF service WSDL
public Object retObject(
  Object objvalue);
}
```

The following illustrates why `xsi:type` is not preserved across an unmarshal/ marshal operation.

- A value of type `uint` is marshalled by WCF serialization mechanism as:
  `<objvalue xsi:type=”xs:unsignedShort”/>`
- JAXB 2.0 unmarshaller unmarshals the value as an instance of `int` and assigns it to parameter `objvalue`.
- The `objvalue` is marshalled back by JAXB 2.0 marshaller with an `xsi:type` of `xs:int`.

  `<objvalue xsi:type=”xs:int”/>`

One way to preserve and roundtrip the `xsi:type` is to use the `mapSimpleTypeDef` customization. The customization makes the mapping of XML Schema Part 2 datatypes one--to-one by generating additional Java classes. Thus, `xs:unsignedShort` will be bound to its own class rather than `int`, as shown:

```
//Java class to which xs:unsignedShort is bound
public class UnsignedShort { ... }
```

The following illustrates how the `xsi:type` is preserved across an unmarshal/ marshal operation:

- A value of type `uint` is marshalled by WCF serialization mechanism as:
  `<objvalue xsi:type=”xs:unsignedShort”/>`
- JAXB 2.0 unmarshaller unmarshals the value as an instance of `UnsignedShort` and assigns it to parameter `objvalue`.
- The `objvalue` is marshalled back by JAXB 2.0 marshaller with an `xsi:type` of `xs:int`.

  `<objvalue xsi:type=”xs:unsignedShort”/>`

Guideline: Use `mapSimpleTypedef` customization where roundtripping of XML Schema types in Table 9–1 are used in `xsi:type`. However, it is preferable to avoid the use of CLR types listed in Table 9–1 since they are specific to .NET platform.

The syntax of the `mapSimpleTypeDef` customization is shown below.

```
<jxb:bindings version="2.0"
         xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
         xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <jaxb:bindings schemaLocation="schema-importedby-wcfsvcwsdl"
               node="/xs:schema">
    <jaxb:globalBindings mapSimpleTypeDef="true"/>
  </jaxb:bindings>
```

Developing a Microsoft .NET Client

This section describes how to develop a .NET client that uses data binding.

Do the following steps to generate a Microsoft .NET client from a Java web service WSDL file:

1. Generate WCF web service client artifacts using the `svcutil.exe` tool:

   ```
   svcutil.exe <java-web-service-wsdl>
   ```

   `svcutil.exe` has the following options for selecting a serializer.

   ```
   svcutil.exe /serializer:auto (default)
   svcutil.exe /serializer:DataContractSerializer
   svcutil.exe /serializer:XmlSerializer
   ```

   We recommend using the default (`/serializer:auto`) option. This option ensures that `svcutil.exe` falls back to `XmlSerializer` if an XML schema construct is used that cannot be processed by `DataContractSerializer`.

   For example, in the following class field `price` is mapped to an XML attribute that cannot be consumed by `DataContractSerializer`.

   ```
   public class POType {
      @javax.xml.bind.annotation.XmlAttribute
      public java.math.BigDecimal price;
   }

   <!-- XML schema fragment -->
   <xs:complexType name="poType">
      <xs:sequence/>
      <xs:attribute name="price" type="xs:decimal"/>
   </xs:complexType>
   ```

2. Develop the .NET client using the generated artifacts.

BP 1.1 Conformance

JAX-WS 2.0 enforces strict Basic Profile 1.1 compliance. The following are known cases where .NET framework does not enforce strict BP 1.1 semantics and their usage can lead to interoperability problems.

BP 1.1 R2211

In `rpclit` mode, BP 1.1 `http://www.ws-i.org/Profiles/BasicProfile-1.1-2006-04-10.html`, R2211 disallows the use of `xsi:nil` in part accessors (see the R2211 for the actual text). From a developer perspective this means that in `rpclit` mode, JAX-WS does not allow a null to be passed in a web service method parameter.

```
//Java Web method
public byte[] retByteArray(byte[] inByteArray)
{
  return inByteArray;
}

<!-- In rpclit mode, the above Java web service method will
throw an exception if the following XML instance with xsi:nil
is passed by a .NET client. -->
<RetByteArray xmlns="http://tempuri.org/">
  <inByteArray a:nil="true" xmlns=""
     xmlns:a="http://www.w3.org/2001/XMLSchema-instance"/>
</RetByteArray>
```

10 Using Atomic Transactions

THIS chapter explains how to configure and use WSIT WS-TX, which implements Web Services-AtomicTransactions (WS-AT) and Web Services-Coordination (WS-Coordination). WSIT WS-TX enables Java EE transactions to work across heterogeneous systems that support WS-AT and WS-Coordination.

About the basicWSTX Example

The basicWSTX example shows the following on the client-side:

1. Developers use existing Java Transaction APIs (JTA). Invocations of transacted web service operations flow transactional context from client to web service.

   Persistent resources updated with client-created transactions are all committed or rolled back as a single atomic transaction.
2. After the client-side code commits or aborts the JTA transaction, the client confirms that all operations in the transaction succeeded or failed via calls to `verify` methods on the transacted web service.

The example also shows the following on the service-side:

1. A transacted web service implemented as a Java servlet

   The "Edit Web Service Attributes" feature in the NetBeans WSIT plug-in is used to configure Transaction Attributes of each web service operation.
2. A transacted web service implemented as Container Managed Transaction (CMT)

   No configuration is necessary for this case.
3. Managed Java EE resources participating in a distributed transaction having its transacted updates all committed or rolled back.

   Both transacted web service operations (servlet and CMT EJB) manipulate these resources.

This example only shows how to use `XATransaction`-enabled JMS. While this example shows WSIT-to-WSIT operations, the client is configured to run on one GlassFish instance and the service runs on the other GlassFish instance. Either the Java client or the Java web service could be replaced by a semantically equivalent Microsoft implementation.

The WS-Coordination/WS-AtomicTransaction protocol messages flow back and forth between the two GlassFish instances just as they would in a Sun-to Microsoft and a Microsoft-to-Sun transaction interoperability scenario.

The `basicWSTX` example was designed so it could be run in either one or in two GlassFish domains. If you run the example in one domain, only one coordinator is used; no WS-Coordination protocol messages will be exchanged. We explain how to run the example in two domains so both protocols, WS-Coordination and WS-AtomicTransaction (WS-AT), are used, as shown in Figure 10–1.

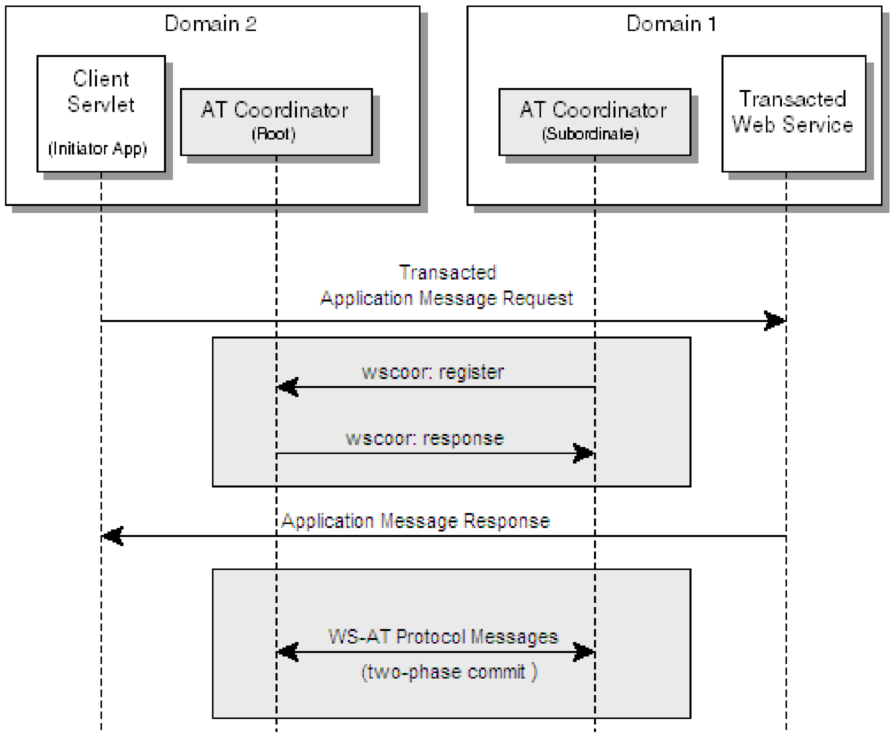


Figure 10–1 WS-Coordination and WS-AtomicTransaction Protocols in Two GlassFish Domains

Figure 10–2 shows the components that make up the example.

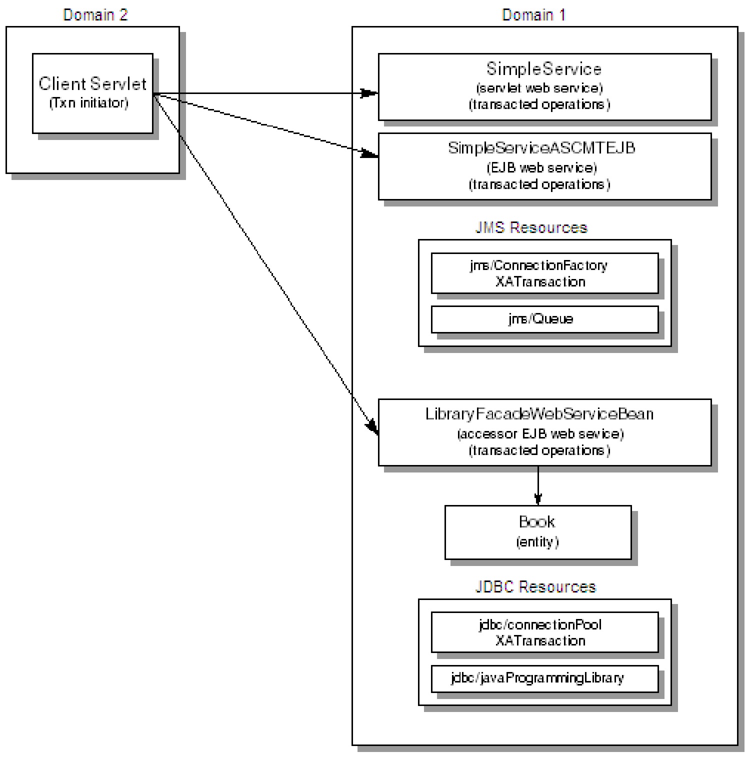


Figure 10–2 Components in the basicWSTX Example

The service, which runs in domain1, is comprised of two components:

- `SimpleService`, a web service that is implemented as a servlet with transacted operations
- `SimpleServiceASCMTEJB`, a container-managed transaction Enterprise bean (CMT EJB) web service

The `SimpleService` web service uses two JMS resources that are created in domain1:

- `jms/ConnectionFactory`, an XATransaction connection factory
- `jms/Queue`, a JMS queue

The client servlet, which runs in domain2, initiates the transaction.

Building, Deploying and Running the basicWSTX Example

Complete the following steps to configure your environment then build, deploy, and run the `basicWSTX` example.

1. Download the sample kit for this example from `https://wsit-docs.dev.java.net/releases/m6/wsittutorial.zip`.
2. Ensure that properties that point to your local GlassFish and WSIT Tutorial installations have been set.
   a. Copy file `<INSTALL>/wsittutorial/examples/bp-project/app-server.properties.sample` to file `<INSTALL>/wsittutorial/examples/bp-project/app-server.properties`.
   b. Set the `javaee.home` and `wsit.tutorial.home` properties in the file `<INSTALL>/wsittutorial/examples/bp-project/app-server.properties`.
   c. Ensure that GlassFish and Ant 1.6.5 or higher have been installed and are on the path. GlassFish includes Ant 1.6.5, which can be found in the `<javaee.home>/lib/ant/bin` directory.
3. Set up your environment to run the basicWSTX example.

   This step performs the following configuration tasks for you:
   - Starts domain1.
   - Creates the resources (`jms/Queue` and `XATransaction jms/ConnectionFactory`) used in the example.
   - Creates and sets up two GlassFish domains.

     The domains can either be created on one machine or on two different machines. We'll show you how to do it on one machine. The first domain, domain1, is created as part of the GlassFish installation.
   - Establishes trust between the two domains by installing each domain's `s1as` security certificate in the other domain's truststore.

   To configure your environment to run the example:

   a. Change to the `<INSTALL>/wsittutorial/examples/wstx/basicWSTX/SampleService` directory:

```
cd <INSTALL>/wsittutorial/examples/wstx/basicWSTX/Sam-
pleService
```

b. Issue the following command to configure your environment to run the example:

```
ant setup
```

4. Register the GlassFish server instances (domain1 and domain2) in the NetBeans IDE.
   a. If the Sun Java System Application Server (domain1) is already registered, go to Step 4g. If it is not, go to Step 4b.
   b. In the Runtime tab, right-click Servers and select Add Server. The Add Server Instance dialog displays.
   c. Choose the server (Sun Java System Application Server) from the pulldown and give it a descriptive name, such as Sun Java System Application Server - domain1 (Server), and then press Next.
   d. Press the Browse button, navigate to the location where the GlassFish server is installed, then press Choose.
   e. Select domain1 from the pulldown, then press Next.
   f. Enter the `admin` account password (`adminadmin`) in the Admin Password field then press Finish. The server instance you just registered is the one in which you will run the web service (`SampleService`).
   g. Right-click Servers and select Add Server. The Add Server Instance dialog displays.
   h. Choose the server (Sun Java System Application Server) from the pulldown and give it a descriptive name, such as Sun Java System Application Server - domain2 (Client), and then press Next.
   i. Press the Browse button, navigate to the location where the GlassFish server is installed, then press Choose.
   j. Select domain2 from the pulldown, then press Next.
   k. Enter the `admin` account password (`adminadmin`) in the Admin Password field then press Finish. The server instance you just registered is the one in which you will run the web service client (`SampleServiceClient`).
5. Associate the SampleService web service with the appropriate instance (domain1) of the GlassFish server.
   a. Select File, then Open Project.

b. Browse to the `<wsit.tutorial.home>/examples/wstx/basicWSTX/` directory, select the SampleService project, and select Open Project Folder.

c. In the Projects tab, right-click SampleService, select Properties, then select the Run category.

d. Use the Server pulldown to point to the Sun Java System Application Server, the default domain, or the Glassfish server instance (domain1) you registered in Step 4.

e. Click OK.

6. Set the proper transaction attributes for each mapping (`wsdl:binding/wsdl:operation`) in the `SimpleService-war` web service.

   This operation creates file `SampleService\SampleService-war\web\WEB-INF\wsit-wstx.sample.service.Simple.xml`, in which the transaction attribute settings for the `SampleService-war` are stored.

   To set the transaction attributes for the `SampleService-war` web service:

   a. In the Projects tab, open the SampleService-war project.

   b. Open the Web Services node.

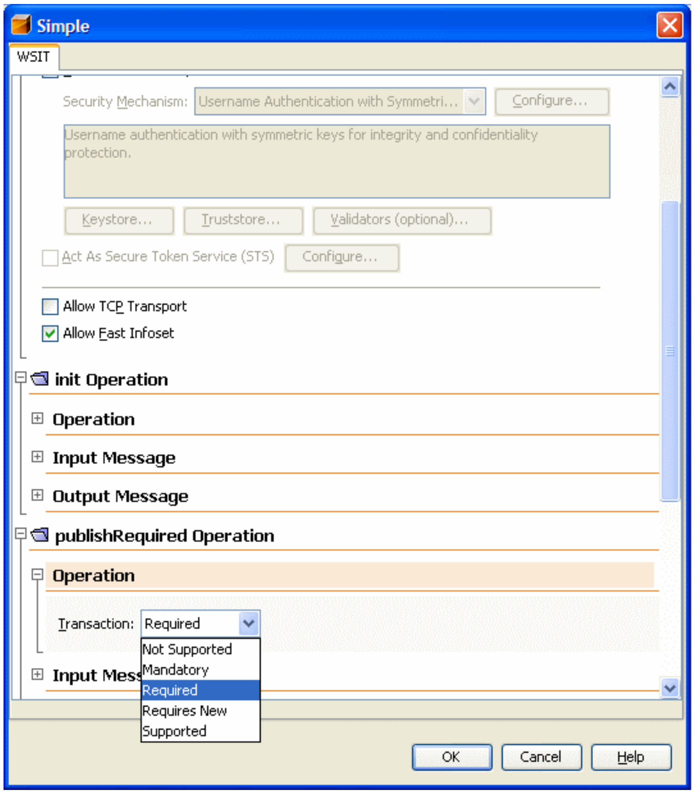
   c. Right-click Simple and select Edit Web Service Attributes.

   d. Select the WSIT tab and open the Operation node and then the method node in each section.

      Select the indicated setting for each of the following operations from the Transaction pulldown:

      - Set `init` to Required
      - Set `publishRequired` to Required
      - Set `publishSupports` to Supported
      - Set `verify` to Required

Figure 10–3 shows how this is done for the `publishRequired` operation.



Security Mechanism: Username Authentication with Symmetri...

Username authentication with symmetric keys for integrity and confidentiality protection.

☐ Allow TCP Transport
☑ Allow Fast Infoset

Transaction: Required

Figure 10–3 Setting the Transaction Attribute for the `publishRequired` Method

e. Click OK.

Transaction attributes for `SampleServiceASCMTEJB` do not need to be set; EJB 3.0 transaction attributes are used.

7. Deploy the SampleService web service.

Right-click SampleService and select Deploy Project. NetBeans will start domain1 and deploy the webservice to that domain.

8. Register the SampleServiceClient client with the appropriate instance (domain2) of the GlassFish server.

a. Select File, then Open Project.

b. Browse to the `<`*`wsit.tutorial.home`*`>/examples/wstx/basicWSTX/` directory, select the SampleServiceClient project, and select Open Project Folder.
c. In the Projects tab, right-click SampleServiceClient, select Properties, then select the Run category.
d. Use the Server pulldown to point to domain2.
e. Click OK.

9. Create web service references for the client (two web service clients, a simpleServlet and a CMT EJB client) and generate the WSDL for both.
   a. In the Projects tab, right-click SampleServiceClient, select New, then select Web Service Client.
   b. Click Browse next to the Project field. The Browse Web Services dialog is displayed.
   c. Open SampleService-war, select Simple, then click OK.
   d. In the Package field, enter `wstx.sample.client`, then click Finish.
   e. Right-click SampleServiceClient, select New, then select Web Service Client.
   f. Click Browse next to the Project field. The Browse Web Services dialog is displayed.
   g. Open SampleService-ejb, select SimpleASCMTEjb, then click OK.
   h. In the Package field, enter `wstx.sample.ejbclient`, then click Finish.

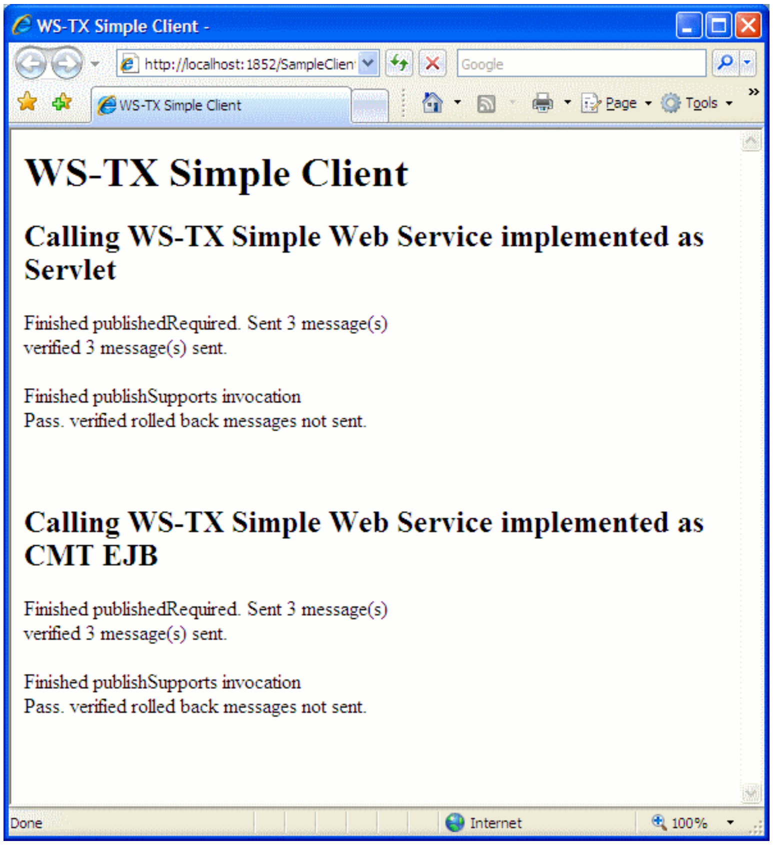
   If transaction attributes for the servlet (see Step 7) or CMT EJB web service have changed, those services must be deployed and client web service references refreshed to get new web service attributes.

   To refresh the client web service references for this example:

   a. In the Projects tab, open the SampleServiceClient, then open Web Service References.
   b. Right-click SimpleService and select Refresh Client to refresh the client node and regenerate the WSDL for the simpleServlet.
   c. Right-click SimpleAsCMTEjb to do the same for the CMT EJB client.

10. Deploy and Run the client.

    Right-click SampleClient and select Run Project. NetBeans will start domain2, deploy the servlet and EJB CMT clients to that domain, then display the results for both in a pop-up browser window, as shown in Figure 10–4.



WS-TX Simple Client

Calling WS-TX Simple Web Service implemented as Servlet

Finished publishedRequired. Sent 3 message(s)
verified 3 message(s) sent.

Finished publishSupports invocation
Pass. verified rolled back messages not sent.

Calling WS-TX Simple Web Service implemented as CMT EJB

Finished publishedRequired. Sent 3 message(s)
verified 3 message(s) sent.

Finished publishSupports invocation
Pass. verified rolled back messages not sent.

Figure 10–4 `basicWSTX` Results

Index