Web Service Guide

JEUS 9

TMAXSOFT

Open Source Software Notice

Document History

Product Version	Guide Version	Date	Remarks
JEUS 9	3.1.2	2025-03-24	-
JEUS 9	3.1.1	2024-12-24	-

Contents

Glossary

In/Out Parameter

The in/out parameter is of a Web service operation used for input and output.

JEUS

Stands for Java Enterprise User Solution, which is a web application server. JEUS version 9 is compatible with the Jakarta EE 9.

Proxy Client

The proxy client is a Web service client created through a stub code which the wsdl2java Ant task created from a WSDL document.

SAAJ

Stands for SOAP message with Attachments API for Java. SAAJ is the standard Java API for processing a SOAP message defined by JCP(Java Community Process).

SOAP

Stands for Simple Object Access Protocol. SOAP is the standard protocol defined by W3C in order to transport XML data on the network, regardless of the programming language and the operating system.

WAS

Web Application Server. Middleware to execute and manage complicated Web applications.

WSDL

Web Service Description Language. WSDL is the standard XML instance defined by W3C in order to describe the Web service interface and implementation.

1. Introduction to Web Services

This chapter introduces the basic web service concepts and standards.

1.1. Basic Concept of a Web Service

A web service is a standardized technology that enables applications to communicate with each other regardless of the platform and programming languages used, and the software interface that implements network access by using the standard XML messaging protocol. In addition, just with an Internet connection, a web service makes businesses available to any user with authority to use the service. For its use, it also defines a convenient environment for the messaging protocol, programming standard, and discovery service.

A web service is an application program that can be accessed through a URI over the Internet. Its interfaces and bindings can be defined and described by XML artifacts. A web service not only communicates with other web services published on the Internet, but it can also communicate with legacy back-end applications.

Until recent years, every application architecture was classified into two major types. One is a single tier mainframe-based system, and the other is a two tier client-server architecture running on a desktop. These architectures are tightly coupled and lack extensibility. Since programs within these architectures are too restricted and difficult to access, they are not useful for users on the web.

This limitation steered the software industry toward Service-Oriented Architecture (SOA), which enables application programs to dynamically communicate with each other on the web. The application programs drove software systems to be divided into modularized sub-systems, and the sub-systems have been implemented using various technologies. They do not need to exist in a single computer, and can also be reused. They are also designed to be accessible to any user on the web through the use of standard web protocols such as XML and HTTP.

SOA is not a new concept. It has been implemented from years ago in many different forms including RMI, COM and CORBA. However, these technologies are either vendor-specific or technology-specific, which prevents them from being universally adapted. Web services have been designed to overcome such shortcomings.

The following are the features of Web services.

- Web services can be invoked over the Internet.
- Web services are self-describing.

  To help web service clients understand the services, a web service publishes descriptions of its roles, functions, and properties.

  Those descriptions are provided in a Web Services Description Language (WSDL) file to allow external applications to be able to locate and utilize the web service.
- Web services communicate with the clients through XML based messages via standard Internet

protocols like HTTP. Web service clients can be any application or another web service.

- The operation type of a web service is either request-response or one-way, and the service is invoked through either synchronous or asynchronous communications. Regardless of the operation type and communication, the basic unit of data exchanged between web services and web service clients is a message.

The following are the advantages of Web services.

- Internet open standard support

  Web services support Internet open standards and are defined by industry agreed standards. As web services communicate based on universally agreed standards such as Simple Object Access Protocol (SOAP), WSDL, and Universal Description, Discovery, and Integration (UDDI), applications supporting web services are not faced with communication problems.

- Platform and language independence
- Application accessibility

  Communication through web protocols like HTTP facilitates access to applications over the current security-heavy network environment with numerous firewalls.

1.2. Web Service Standards

With the release of Java 2 Platform, Enterprise Edition (Java EE) specification version 1.4, web services became a significant part of Java EE standard. Services including transaction support, database connection, and life cycle management can now be supported without modifying the program source code.

To provide better web services, new specifications have been added to Java EE version 5, and the functions of existing specifications have been re-organized and improved.

The following are the standards that the current JEUS web services must conform to:

- Implementing Enterprise Web Services (EWS)
- Java API for XML-based Web Services (JAX-WS)
- Java Architecture for XML Binding (JAXB)
- SOAP with Attachments API for Java (SAAJ)
- Streaming API for XML
- Web Service Metadata for the Java Platform
- XML, XML Namespace, XML Infoset, XML Schema
- SOAP, MTOM, WS-Addressing
- WS-ReliableMessaging, WS-Coordination, WS-AtomicTransaction
- WSDL, WS-Policy, WS-MetadataExchange
- WS-Security Policy, WS-Security, WS-Trust, WS-SecureConversation

The following sections describe some of the most commonly used standards among those listed above.

1.2.1. SOAP Standard

SOAP, which stands for Simple Object Access Protocol, is a lightweight, XML-formatted protocol used for exchanging information in distributed systems.

SOAP supports the following communication protocol.

- Remote Procedure Call (RPC) protocol

  RPC-style communication involves a request-response mechanism, where the client sends procedure-oriented messages to a service endpoint. The service endpoint processes these messages and returns appropriately constructed response messages.

- Message-oriented communication

  In message-oriented communication, the primary focus is on sending and receiving well-structured messages that may not necessarily correspond directly to remote method calls. This approach allows for processing service requests from the sender (client) without an immediate response requirement. Message-oriented communication is particularly useful for exchanging various formats of documents in business scenarios, making it well-suited for scenarios where data exchange takes precedence over immediate interactions. Due to its emphasis on exchanging documents, this communication style is often referred to as document-style communication.

The SOAP protocol consists of the following elements.

- SOAP message envelope

  The SOAP message envelope contains information about how messages are processed, who processes them, and it also envelops the body of the message.

- Encoding rule for describing application data type objects
- Other requirements for representing remote procedure calls and responses

The following describes the characteristics of SOAP.

- Protocol neutrality
- Language independence
- Platform and OS independence
- SOAP XML messages can include additional content or attachments by using standards like MIME.

1.2.2. WSDL Standard

The Web Service Description Language (WSDL) is used to describe RPC-style and message-oriented network services in XML format. Development tools, service developers, or deployers (service providers) can create WSDL files. If the files are opened on the Internet, the service is exposed to other clients which creates an environment where the clients can also access the service.

Client-side programmers and service consumers can use the published WSDL descriptions to obtain information about the available web services. Based on the information, they can create proxies or program templates to access the services.

1.2.3. UDDI Standard

The Universal Description, Discovery, and Integration (UDDI) specification defines a standard way to open and search for web service related information. A service provider saves its services in the UDDI, an online repository that is already known to the service consumers, and the consumers access the repository to search for desired list of services.

1.2.4. JAX-WS, JAXB, StAX Standards

Java API for XML-based Web Services (JAX-WS) standard defines the web service API provided by Sun Microsystems. As the successor to JAX-RPC standard, JAX-WS is a new enhanced integration of JAXB standard for XML binding, StAX standard for standard streaming parsers, and improved SAAJ standard. By using JAX_WS, all of the descriptor files, which have been requested by Java EE 1.4 specification, can be replaced by the annotation function of Java SE 5.

1.2.5. SAAJ Standard

SOAP with Attachments API for Java (SAAJ) standard provides a programming interface with direct access to the SOAP protocol. Any user, who is familiar with processing low-level protocols, can create a SOAP message without difficulty.

1.3. SOAP Message Exchange and Encoding

SOAP supports RPC and document-oriented methods for exchanging messages. SOAP uses the encoding mechanism configured as the EncodingStyle attribute for the SOAP message element.

This section describes the characteristics of a SOAP message.

- SOAP Message Composition

  A SOAP message has a SOAP envelope, and the envelope has two sub-elements: header and body. The header is optional, and its sub-elements, header blocks, may be seen as logical sets of data. The SOAP body, an essential element of a SOAP message, contains the contents of the actual message that needs to be delivered.

- SOAP Message Encoding

SOAP supports both RPC-style and document-oriented (message-oriented) communications.

- RPC style communication

  RPC style communication means exchanging messages through a remote procedure call. Client and server programs should be implemented with well defined programming model. The client calls a method with arguments, and the server returns a value as a response. For RPC-oriented exchanges, SOAP defines a separate body structure for the message.

- Document-oriented communication

  In document-oriented communication, XML documents are exchanged and the definition of the XML elements are left to be interpreted by the server and client. Thus, SOAP does not define any message body structure for document-oriented exchanges. The XML message structure is defined by a separate XML schema or by the application itself.

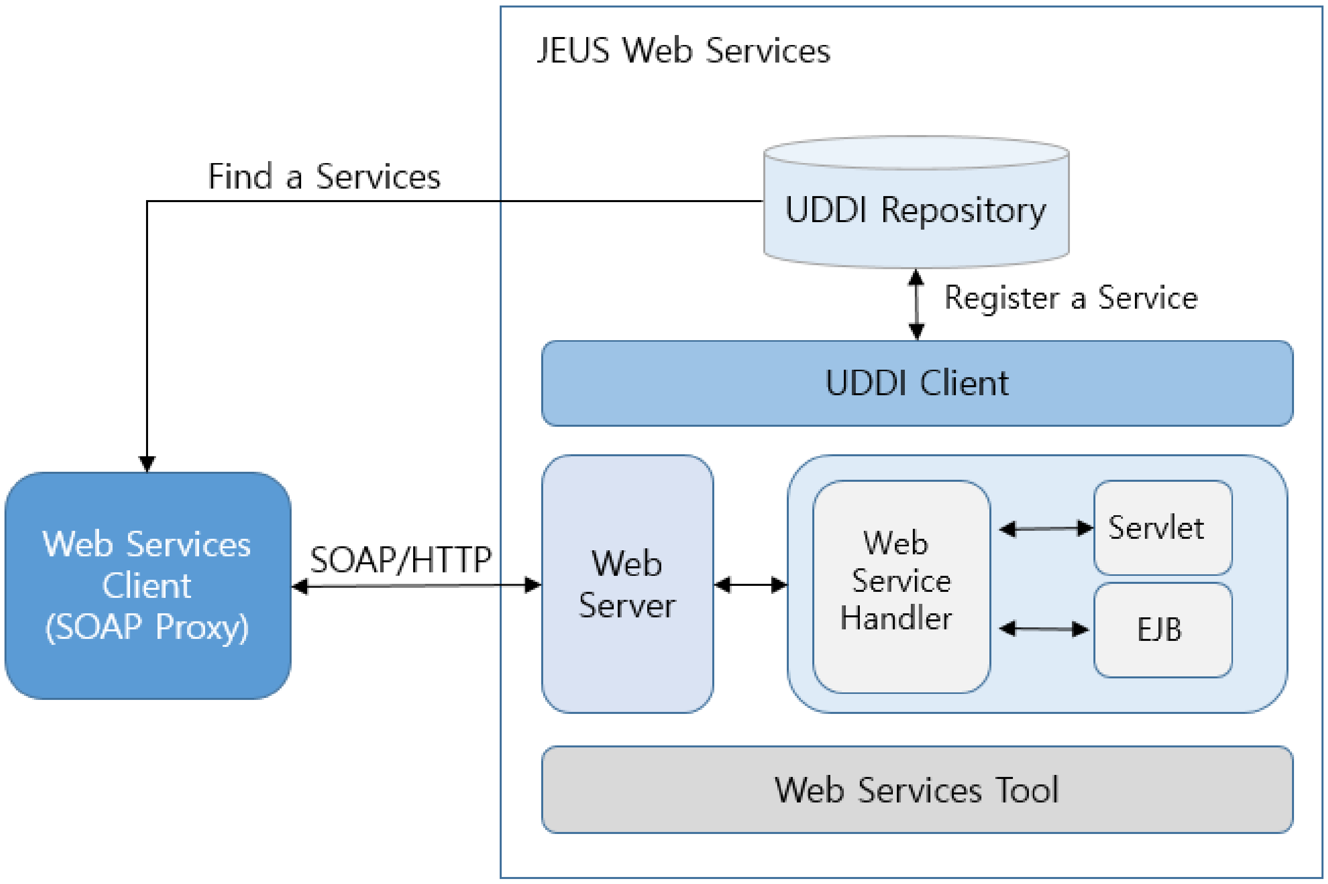
2. JEUS Web Services

This chapter describes the concepts of JEUS web services and supported specifications. It also introduces the configuration files, tools, and system variables for JEUS web services.

2.1. Basic Architecture

JEUS 9, which adheres to Jakarta EE 8, supports JAX-WS. The most useful feature of JAX-WS is the implementation of POJO (Plain Old Java Object) web services. JEUS web services guarantee compatibility with vendors that are Jakarta EE 8-compliant.

The following figure shows the structure of JEUS web services.



The structure of JEUS web services

- Annotations and XML Configuration Files

  JEUS web services use EJB or Java class for its business logic. An EJB runs inside the EJB container and Java classes run inside the web engine.

  By using the annotations provided by Java SE 5, JEUS web services support the implementation of JAX-WS web services that are used to configure web service related settings.

- Tools and Utilities

  JEUS web services use the command line tool and Apache Ant tool to create and manage WSDL files and Java classes. The following sections explain how to use these tools.

Apache Ant 1.6.1 or later must be installed in order to use the Ant tool functions. Apache Ant can be downloaded from http://ant.apache.org.

- System Environment Variables

  There are no special system environment variables for JEUS web services. Those described in "JEUS Web Engine Guide" and "JEUS EJB Guide" are sufficient for operating with JEUS web services.

2.2. Designing Web Services

This section describes considerations for designing web services.

2.2.1. Selecting a Web Service Backend

Web service components are Jakarta EE components that run in a web engine or an EJB container.

The following are the entities that can be published as a web service backend.

- Java Class File

  A Java class file runs in a web container. It’s faster and easier to process tasks using Java classes than by creating an EJB.

  Java class web service backend is a good choice when persistence, security, and transactions that are provided by an EJB container are not required.

- Stateless Session EJB

  An EJB is a programming model, which is good for any transaction-oriented systems, where a stateless session EJB runs in an EJB container.

  An EJB is a good choice for implementing the business logic for efficiently managing transactions in those systems that require frequent DB updates or deletions.

  An EJB backend can be a good choice as the web service backend even for non-transaction-oriented systems. A Container Managed Persistence (CMP) Bean is a good example of this. For more information regarding CMP bean, refer to JEUS Enterprise JavaBeans Guide.

  As already stated, the existing business logic may be implemented as an EJB as needed (stateless session bean or CMP). If the business logic is already implemented as a stateless session bean, it is recommended to use stateless session bean web service backend to expose the existing EJB logic as a web service.

  Since EJB web services usually allow only stateless session beans as a web service backend, CMP may seem not useful for web services. However, CMP business logic can be exposed as web services. This can be done by implementing a stateless session bean as the business logic interface for the CMP bean and exposing the stateless session bean as a web service. A stateless session bean web service backend can be a good choice because it provides the strength of the existing CMP business logic with added web service functionality.

2.2.2. Document-oriented Communication

JEUS web services support document-oriented information exchange.

In document-oriented communication, when XML documents are exchanged, the content is interpreted by the client and server. Therefore, the restrictions on the body structure of the SOAP messages are not defined by SOAP. Instead, an application program or a separate XML schema determines the structure of an XML document of the SOAP body.

JEUS web services provide 2 types of document communications: standard document and document-wrapped.

Type	Description
Standard document-oriented	Standard document-oriented web service operations take only one parameter, which is XML document.
Document-wrapped	Document-wrapped web service operations can take any number of parameters, but the parameter values are wrapped into one complex data type in the SOAP message.

2.2.3. Implementing Web Services

Typically, implementing a web service can start out with a service endpoint or WSDL. Since each path has its own advantages, a developer can decide which path to take. They can be implementation of JAX-WS web service.

- Starting with Web Service Endpoint

  Services can be implemented in a simple, easy, and intuitive way.

- Starting with WSDL

  This method is suitable for implementing a language-independent services focused on interoperability.

Starting with Web Service Endpoint

By starting out with creating a web service endpoint, developers can generate a WSDL file from it. This method has the advantages of an easy and intuitive implementation process. This enables a developer to develop service endpoint and implementation in a familiar environment without having to worry about interoperability with other platforms in a Java environment. The developer can then derive a platform independent WSDL from the Java codes.

JAX-WS provides guidelines on how to derive WSDL from service endpoints so that WSDL can be platform independent. This is the easiest method for developing web services for JAVA developers.

The only thing to note about JAX-WS is that it does not require descriptions about service configurations. According to JAX-WS specification, a web service can be implemented using the wsgen toolkit without a service configuration file.

Web service implementation using the JAX-WS method is further discussed in Implementing JEUS Web Services.

Starting with WSDL

First create a WSDL file, and then generate web service Implementation from it.

Core objective of a web service is its interoperability with different platforms. This method is appropriate for generating interoperable service that is independent of the programming language. The platform's characteristics will be reflected in the WSDL generated from a web service implementation, leaning towards Java. The descriptions derived from a WSDL, on the other hand, are more neutral than the XML types and terms used for web service descriptions. This allows developers who are not familiar with the Java language to create a WSDL using general nomenclature.

However, it will require developers to have a deeper understanding of WSDL. JAX-WS web service implementation is further described in Implementing JEUS Web Services.

2.2.4. Creating SOAP Message Handlers

A message handler needs to be created in order to directly handle the SOAP message's header, body, or attachment.

The handler framework can be used for JAX_WS. For more information, refer to Handler Framework.

Part I. JEUS 9 Web Services

JEUS 9 web services support JAX-WS web service which is the Jakarta EE 9 web service standard. The most important feature of JAX-WS web service is the ability to freely implement POJO web services without configuration files. The POJO web service is implemented using the annotation functionality of Java SE 5.

This document discusses how to implement the JAX-WS web service and run the service on JEUS, and how to create a client which invokes JAX-WS web service and handle the declaration of custom binding, handlers, and messages at the XML level. It also describes how to transmit a message more efficiently through asynchronous and MIME attachments, how to implement faster web services, and finally how to handle XML in various ways.

The following chapters explain how JEUS 9 web services support JAX-WS web services.

3. Implementing JEUS Web Services

This chapter describes how to implement JEUS Web services based on Java classes, EJBs, and WSDL.

3.1. Overview

JEUS 9 web services support **JAX-WS** web service, a new web service standard of Jakarta EE 9.

Like JAXB (standard for data binding between XML documents and Java objects) and SAAJ (standard API to implement and modify XML SOAP messages. While in JAX-RPC it was used by a developer to implement a message handler, in JAX-WS, a basic framework for message handlers is provided, and SAAJ works behind it), JAX-WS is a core component of web services, and has been developed to replace the JAX-RPC web service.

Looking at its history, since JAX-RPC was released before JAXB was fully developed, it included a built-in data binding function, which is a function provided by JAXB. However, it became increasingly difficult for JAX-RPC web service to maintain many added functions to comply with XML standards. As JAXB provided more enhanced data binding function as its focus, there was no longer a need for JAX-RPC to keep its data binding functionality. Thus, general web service functions are now managed by JAX-WS, and the data binding function is managed separately by JAXB. JAXB will be discussed later.

Basic description for JAX-WS is provided in JSR 224 (http://jcp.org/jsr/detail/224.jsp). The annotation functionality of Java SE 5 enabled JAX-WS to process various web service deployment descriptor (DD) files using annotations, and to implement Plain Old Java Object (POJO) web services.

In this chapter, a JAX-WS web service is implemented from a Java class file and WSDL. To implement a web service from a Java class, a developer can create portable artifacts through annotations just with a service endpoint implementation class using the **wsgen** tool. To implement a web service from WSDL, the developer can create a service endpoint interface and portable artifacts using the **wsimport** tool with the created WSDL file, and then create a Java class to implement them.

3.2. Implementing Web Services from Java Classes

The following are the rules that JAX-WS requires to implement web services from a Java endpoint implementation class.

- Must include the jakarta.jws.WebService annotation.
- Methods can include the jakarta.jws.WebMethod annotation.
- All methods can throw the java.rmi.RemoteException exception along with service-specific exceptions.
- Parameters and return types of all methods must be stated in JAXB definition of Java to XML schema mapping.
- A parameter or return type of a method must not implement the java.rmi.Remote interface directly or indirectly.

The following is an example of a simple Java endpoint implementation class in accordance with the aforementioned rules.

Java Class Web Service: <Addnumbersimpl.java>

```
package fromjava.server;

@WebService
public class AddNumbersImpl {

    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

An annotation, @WebService, is included in the previous example. When the annotation, @WebService, is configured and an endpoint implementation logic is created, the server-side implementation of the web service is complete. Next, portable artifacts must be created using a tool provided by JEUS 9 web services. The portable artifacts consist of several Java bean classes, which are used to convert method calls or responses into Java objects or XML documents, and service specific exception classes. Use **wsgen** to create the portable artifacts.

The following is an example of using **wsgen**.

```
$ wsgen -help

Usage: WSGEN [options] <SEI>

where [options] include:
  -classpath <path>          specify where to find input class files
  -cp <path>                 same as -classpath <path>
  -d <directory>             specify where to place generated output files
  -extension                 allow vendor extensions - functionality
                             not specified by the specification.
                             Use of extensions may result in
                             applications that are not portable or
                             may not interoperate with other implementations
  -help                      display help
  -keep                      keep generated files
  -r <directory>             resource destination directory, specify
                             where to place resouce files such as WSDLs
  -s <directory>             specify where to place generated source files
  -verbose                   output messages about what the compiler is doing
  -version                   print version information
  -wsdl[:protocol]           generate a WSDL file. The protocol is optional.
                             Valid protocols are [soap1.1, Xsoap1.2],
                             the default is soap1.1.
                             The non stanadard protocols [Xsoap1.2]
                             can only be used in conjunction with the
                             -extension option.
  -servicename <name>        specify the Service name to use in the generated
                             WSDL Used in conjunction with the -wsdl option.
  -portname <name>           specify the Port name to use in the generated
                             WSDL Used in conjunction with the -wsdl option.
```

```
Examples:
  wsgen -cp . example.Stock
  wsgen -cp . example.Stock -wsdl
        -servicename {http://mynamespace}MyService
```

JEUS 9 web services support an ANT TASK for the wsgen tool. For more information about console commands and ANT TASK, refer to "wsgen" console tool and "wsgen" plugin in *JEUS Reference Guide*.

As shown in the following example, portable artifacts are created by using the Java endpoint implementation class that is implemented with the wsgen tool.

Java Class Web Service: <build.xml>

```
...
<target name="build_server" depends="init">
    <antcall target="do-compile">
        <param name="javac.excludes" value="fromjava/client/" />
    </antcall>
    <antcall target="wsgen">
        <param name="sib.file" value="fromjava.server.AddNumbersImpl" />
    </antcall>
    <antcall target="do-package-war" />
</target>
...
```

When portable artifacts and WSDL file are generated from the previous Java endpoint implementation class, AddnumbersImpl, by using the wsgen tool, the result is displayed as in the following.

```
AddNumbersImplService.wsdl
AddNumbersImplService_schema1.xsd
fromjava/server/jaxws/AddNumbers.class
fromjava/server/jaxws/AddNumbersResponse.class
```

AddNumbers and AddNumbersResponse files are Java beans used by JAXB to convert a request and response of the addNumbers method into a Java object or XML document.

The AddNumbersImplService.wsdl file is the WSDL file of the web service, and the AddNumbersImplService_ schema1.xsd schema file contains the data type definitions used by the web service within the WSDL document.

The following are the contents of the WSDL file.

Java Class Web Service: <AddNumbersImplService.wsdl>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://server.fromjava/"
    name="AddNumbersImplService" xmlns:tns="http://server.fromjava/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://server.fromjava/"
                schemaLocation="AddNumbersImplService_schema1.xsd" />
        </xsd:schema>
    </types>
    <message name="addNumbers">
        <part name="parameters" element="tns:addNumbers" />
    </message>
    <message name="addNumbersResponse">
        <part name="parameters" element="tns:addNumbersResponse" />
    </message>
    <portType name="AddNumbersImpl">
        <operation name="addNumbers">
            <input message="tns:addNumbers" />
            <output message="tns:addNumbersResponse" />
        </operation>
    </portType>
    <binding name="AddNumbersImplPortBinding" type="tns:AddNumbersImpl">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
            style="document" />
        <operation name="addNumbers">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>
    <service name="AddNumbersImplService">
        <port name="AddNumbersImplPort"
            binding="tns:AddNumbersImplPortBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL" />
        </port>
    </service>
</definitions>
```

As shown in the previous example, server-side portable artifacts are created for <message> elements of a WSDL file. The portable artifacts are responsible for transmitting messages between a JAX-WS engine and endpoint class. Portable artifact Java classes of the two <message> elements are AddNumbers and AddNumbersResponse.

The following shows a Java bean class for AddNumbers <message> element that is invoked by a client.

Java Class Web Service: <AddNumbers.java>

```
@XmlRootElement(name = "addNumbers", namespace = "http://server.fromjava/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "addNumbers", namespace = "http://server.fromjava/",
         propOrder = {"arg0", "arg1" })
public class AddNumbers {
    @XmlElement(name = "arg0", namespace = "")
    private int arg0;

    @XmlElement(name = "arg1", namespace = "")
    private int arg1;

    public int getArg0() {
        return this.arg0;
    }

    public void setArg0(int arg0) {
        this.arg0 = arg0;
    }

    public int getArg1() {
        return this.arg1;
    }

    public void setArg1(int arg1) {
        this.arg1 = arg1;
    }
}
```

The following is an example of a Java bean class for the AddNumbersResponse <message> element that is returned by the service.

Java Class Web Service: <AddNumbersResponse.java>

```
@XmlRootElement(name = "addNumbersResponse", namespace = "http://server.fromjava/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "addNumbersResponse", namespace = "http://server.fromjava/")
public class AddNumbersResponse {

    @XmlElement(name = "return", namespace = "")
    private int _return;

    public int get_return() {
        return this._return;
    }

    public void set_return(int _return) {
        this._return = _return;
    }
}
```

3.3. Implementing EJB Web Services

This section discusses EJB web service programming model for implementing web services by using stateless session beans running on an EJB container.

EJB 4.0 programming model in JEUS 9 provides many functions to make it easier to create a service endpoint implementation class. For instance, it does not implement the jakarta.ejb.SessionBean or jakarta.ejb.EntityBean interface any longer, but includes it as an annotation. Also, its session bean no longer has to implement the component or home interface.

For more information about those functionalities provided by JEUS 9, refer to "JEUS EJB Guide".

To implement bean classes using the EJB 4.0 functionality provided by JEUS 9, and use them as JAX-WS endpoint implementation classes, include the @WebService annotation in the bean class. The following is an example of a simple EJB endpoint implementation class.

Implementing an EJB Web Service: <Addnumbersimpl.java>

```
package fromejb.server;

@Stateless
@WebService
public class AddNumbersImpl {

    public AddNumbersImpl() {

    }

    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

Note that the annotations, @Stateless and @WebService, are added to the previous example. Once the @WebService annotation is configured and an endpoint implementation logic is created, the server-side implementation of the web service is complete.

Next, portable artifacts must be created by using a tool provided by JEUS 9 web service. The portable artifacts consist of service specific exception classes and several Java bean classes that are used to convert method calls or responses into Java objects or XML documents. Use **wsgen** to create the portable artifacts.

The following shows how to use **wsgen**.

```
$ wsgen -help

Usage: WSGEN [options] <SEI>

where [options] include:
  -classpath <path>          specify where to find input class files
  -cp <path>                 same as -classpath <path>
  -d <directory>             specify where to place generated output files
  -extension                 allow vendor extensions - functionality
                             not specified by the specification.
                             Use of extensions may result in
                             applications that are not portable or
                             may not interoperate with other implementations
  -help                      display help
  -keep                         keep generated files
  -r <directory>                resource destination directory, specify
                                where to place resouce files such as WSDLs
  -s <directory>                specify where to place generated source files
  -verbose                      output messages about what the compiler is doing
  -version                      print version information
  -wsdl[:protocol]              generate a WSDL file. The protocol is optional.
                                Valid protocols are [soap1.1, Xsoap1.2],
                                the default is soap1.1.
                                The non stanadard protocols [Xsoap1.2]
                                can only be used in conjunction with the
                                -extension option.
  -servicename <name>           specify the Service name to use in the generated
                                WSDL Used in conjunction with the -wsdl option.
  -portname <name>              specify the Port name to use in the generated
                                WSDL Used in conjunction with the -wsdl option.

Examples:
  wsgen -cp . example.Stock
  wsgen -cp . example.Stock -wsdl
        -servicename {http://mynamespace}MyService
```

JEUS 9 web services support an ANT TASK for the wsgen tool. For more information about console commands and ANT TASK, refer to "wsgen" console tool and "wsgen" plugin in *JEUS Reference Guide*.

As shown in the following example, portable artifacts are created by using the Java endpoint implementation class that is implemented with the wsgen tool.

Implementing an EJB Web Service: <build.xml>

```
...

<target name="build_server" depends="init">
    <antcall target="do-compile">
        <param name="javac.excludes" value="fromejb/client/" />
    </antcall>
    <antcall target="wsgen">
        <param name="sib.file" value="fromejb.server.AddNumbersImpl" />
    </antcall>
    <antcall target="do-package-jar" />
</target>

...
```

The following is the result of using the wsgen tool to generate portable artifacts and WSDL file by using the previous AddnumbersImpl Java endpoint implementation class.

```
AddNumbersImplService.wsdl
AddNumbersImplService_schema1.xsd
fromjava/server/jaxws/AddNumbers.class
fromjava/server/jaxws/AddNumbersResponse.class
```

AddNumbers and AddNumbersResponse files are Java beans used by JAXB to convert a request and response of the addNumbers method into a Java object or XML document.

The AddNumbersImplService.wsdl file is the WSDL file of the web service, and the AddNumbersImplService_ schema1.xsd schema file contains the data type definitions used by the web service within the WSDL document.

The following are the contents of the WSDL file.

Implementing an EJB Web Service: <AddNumbersImplService.wsdl>

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://server.fromejb/"
    name="AddNumbersImplService" xmlns:tns="http://server.fromejb/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://server.fromejb/"
                schemaLocation="AddNumbersImplService_schema1.xsd" />
        </xsd:schema>
    </types>
    <message name="addNumbers">
        <part name="parameters" element="tns:addNumbers" />
    </message>
    <message name="addNumbersResponse">
        <part name="parameters" element="tns:addNumbersResponse" />
    </message>
    <portType name="AddNumbersImpl">
        <operation name="addNumbers">
            <input message="tns:addNumbers" />
            <output message="tns:addNumbersResponse" />
        </operation>
    </portType>
    <binding name="AddNumbersImplPortBinding" type="tns:AddNumbersImpl">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
            style="document" />
        <operation name="addNumbers">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>
    <service name="AddNumbersImplService">
        <port name="AddNumbersImplPort"
            binding="tns:AddNumbersImplPortBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL" />
        </port>
    </service>
</definitions>
```

As shown in the previous example, server-side portable artifacts are created for <message> elements of a WSDL file. The portable artifacts are responsible for transmitting messages between a JAX-WS engine and endpoint class.

The following shows a Java bean class for AddNumbers <message> element that is invoked by a client.

Implementing an EJB Web Service: <AddNumbers.java>

```
@XmlRootElement(name = "addNumbers", namespace = "http://server.fromejb/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "addNumbers", namespace = "http://server.fromejb/", propOrder = {
        "arg0", "arg1" })
public class AddNumbers {

    @XmlElement(name = "arg0", namespace = "")
    private int arg0;

    @XmlElement(name = "arg1", namespace = "")
    private int arg1;

    public int getArg0() {
        return this.arg0;
    }

    public void setArg0(int arg0) {
        this.arg0 = arg0;
    }

    public int getArg1() {
        return this.arg1;
    }

    public void setArg1(int arg1) {
        this.arg1 = arg1;
    }
}
```

The following is an example of a Java bean class for the AddNumbersResponse <message> element that is returned by the service.

Implementing an EJB Web Service: <AddNumbersResponse.java>

```
@XmlRootElement(name = "addNumbersResponse", namespace = "http://server.fromejb/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "addNumbersResponse", namespace = "http://server.fromejb/")
public class AddNumbersResponse {

    @XmlElement(name = "return", namespace = "")
    private int _return;

    public int get_return() {
        return this._return;
    }

    public void set_return(int _return) {
        this._return = _return;
    }
}
```

3.4. Implementing Web Services from WSDL

To implement web services from WSDL, create a service endpoint interface by using the **wsimport** tool, provided by JEUS 9 web services.

The following is an example of a WSDL file.

Implementing a WSDL Web Service: <AddNumbers.wsdl>

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions name="AddNumbers" targetNamespace="urn:AddNumbers"
             xmlns:impl="urn:AddNumbers"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
    <types>
        <xsd:schema elementFormDefault="qualified"
                    targetNamespace="urn:AddNumbers"
                    xmlns="http://www.w3.org/2001/XMLSchema">
            <complexType name="addNumbersResponse">
                <sequence>
                    <element name="return" type="xsd:int" />
                </sequence>
            </complexType>
            <element name="addNumbersResponse"
                     type="impl:addNumbersResponse" />
            <complexType name="addNumbers">
                <sequence>
                    <element name="arg0" type="xsd:int" />
                    <element name="arg1" type="xsd:int" />
                </sequence>
            </complexType>
            <element name="addNumbers" type="impl:addNumbers" />
        </xsd:schema>
    </types>

    <message name="addNumbers">
        <part name="parameters" element="impl:addNumbers" />
    </message>
    <message name="addNumbersResponse">
        <part name="result" element="impl:addNumbersResponse" />
    </message>

    <portType name="AddNumbersPortType">
        <operation name="addNumbers">
            <input message="impl:addNumbers" name="add" />
            <output message="impl:addNumbersResponse"
                    name="addResponse" />
        </operation>
    </portType>

    <binding name="AddNumbersBinding"
             type="impl:AddNumbersPortType">
        <soap:binding
            transport="http://schemas.xmlsoap.org/soap/http"
            style="document" />
        <operation name="addNumbers">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>

    <service name="AddNumbersService">
        <port name="AddNumbersPort" binding="impl:AddNumbersBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL" />
        </port>
    </service>
</definitions>
```

After implementing a WSDL file, create a service endpoint interface and portable artifacts by using **wsimport**.

The following shows how to use **wsimport**.

```
$ wsimport -help

Usage: wsimport [options] <WSDL_URI>

where [options] include:
  -b <path>                 specify jaxws/jaxb binding files or additional schemas
                            (Each <path> must have its own -b)
  -B<jaxbOption>           Pass this option to JAXB schema compiler
  -catalog <file>           specify catalog file to resolve external
                            entity references supports TR9401,
                            XCatalog, and OASIS XML Catalog format.
  -d <directory>            specify where to place generated output files
  -extension                allow vendor extensions - functionality
                            not specified by the specification. Use
                            of extensions may result in applications
                            that are not portable or may not
                            interoperate with other implementations
  -help                     display help
  -httpproxy:<host>:<port>  specify a HTTP proxy server (port defaultsto 8080)
  -keep                     keep generated files
  -p <pkg>                  specifies the target package
  -quiet                    suppress wsimport output
  -s <directory>            specify where to place generated source files
  -target <version>         enerate code as per the given JAXWS spec version
                            e.g. 2.0 will generate compliant code for JAXWS 2.0 spec
  -verbose                  output messages about what the compiler is doing
  -version                  print version information
  -wsdllocation <location>  @WebServiceClient.wsdlLocation value

Examples:
  wsimport stock.wsdl -b stock.xml -b stock.xjb
wsimport -d generated http://example.org/stock?wsdl
```

JEUS 9 web services support the ANT TASK for the wsimport tool. For more information about console commands and ANT TASK, refer to "wsimport" console tool and "wsimport" plugin in *JEUS Reference Guide*.

As shown in the following example, a service endpoint interface and portable artifacts are created using the WSDL file, which was implemented using the wsimport tool.

Implementing a WSDL Web Service: <build.xml>

```
...

<target name="build_server" depends="init">
    <mkdir dir="${build.classes.dir}" />
    <antcall target="wsimport">
        <param name="package.name" value="fromwsdl.server" />
        <param name="binding.file" value="" />
        <param name="wsdl.file" value="${src.web}/WEB-INF/wsdl/AddNumbers.wsdl" />
    </antcall>
    <antcall target="do-compile">
        <param name="javac.excludes" value="fromwsdl/client/" />
    </antcall>
    <antcall target="do-package-war" />
</target>

...
```

The following is the service endpoint interface and portable artifacts that are created by using the wsimport ANT TASK.

```
fromwsdl/server/AddNumbers.class
fromwsdl/server/AddNumbersPortType.class
fromwsdl/server/AddNumbersResponse.class
fromwsdl/server/AddNumbersService.class
fromwsdl/server/ObjectFactory.class
fromwsdl/server/package-info.class
```

AddNumbers and AddNumbersResponse files are Java beans used by JAXB to convert a request and response of the addNumbers method into a Java object or XML document.

The AddNumbersPortType file is the service endpoint interface, and the AddNumbersService file is a Java class used by the client as a proxy. The ObjectFactory and package-info classes are files created by JAXB.

The following is the content of the file. First, AddNumbersPortType class, which is the service endpoint interface obtained from the WSDL file by using the **wsimport** tool, is implemented.

Implementing a WSDL Web Service: <AddNumbersPortType.java>

```
@WebService(name = "AddNumbersPortType", targetNamespace = "urn:AddNumbers")
@XmlSeeAlso( { ObjectFactory.class })
public interface AddNumbersPortType {
    @WebMethod
    @WebResult(targetNamespace = "urn:AddNumbers")
    @RequestWrapper(localName = "addNumbers",
        targetNamespace = "urn:AddNumbers",
        className = "fromwsdl.server.AddNumbers")
    @ResponseWrapper(localName = "addNumbersResponse",
        targetNamespace = "urn:AddNumbers",
        className = "fromwsdl.server.AddNumbersResponse")
    public int addNumbers(
            @WebParam(name = "arg0", targetNamespace = "urn:AddNumbers")
            int arg0,
            @WebParam(name = "arg1", targetNamespace = "urn:AddNumbers")
            int arg1);
}
```

The service endpoint interface contains annotations required during runtime or for dynamic binding. Such annotations are required to convert XML documents to Java objects, or Java objects to XML documents.

The service endpoint interface can be used to create WSDL and schema files. However, the created WSDL and schema files may not be identical to the original WSDL and schema files which were used to obtain the service endpoint interface.

The following is the AddNumbers and AddNumbersResponse classes, which are Java bean classes used by JAXB to convert a request and response of the addNumbers method into a Java object or XML document.

Implementing a WSDL Web Service: <AddNumbers.java>

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "addNumbers", propOrder = { "arg0", "arg1" })
public class AddNumbers {

    protected int arg0;

    protected int arg1;

    public int getArg0() {
        return arg0;
    }
    public void setArg0(int value) {
        this.arg0 = value;
    }
    public int getArg1() {
        return arg1;
    }
    public void setArg1(int value) {
        this.arg1 = value;
    }
}
```

Implementing a WSDL Web Service: <AddNumbersResponse.java>

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "addNumbersResponse", propOrder = { "_return" })
public class AddNumbersResponse {

    @XmlElement(name = "return")
    protected int _return;

    public int getReturn() {
        return _return;
    }
    public void setReturn(int value) {
        this._return = value;
    }
}
```

These two Java bean classes correspond to the <message> elements, AddNumbers and AddNumbersResponse, of the previously created WSDL file.

Next, a service endpoint implementation class, which includes the business logic of the service, is created using the service endpoint interface and portable artifacts created by WSDL. To create the class, an annotation indicating the name of the service endpoint interface must be provided in the Java class implementation file. Specify the path of the WSDL file, the target namespace of the WSDL file, and the service and port names as in the following.

Implementing a WSDL Web Service: <AddNumbersImpl.java>

```
package fromwsdl.server;

@jakarta.jws.WebService(
endpointInterface = "fromwsdl.server.AddNumbersPortType",
    wsdlLocation="WEB-INF/wsdl/AddNumbers.wsdl",
    targetNamespace = "urn:AddNumbers",
    serviceName = "AddNumbersService",
    portName = "AddNumbersPort"
)
public class AddNumbersImpl {
    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

As shown in the previous example, the name of the service endpoint interface in the Java class implementation is specified as endpointInterface, wsdlLocation, targetNamespace, serviceName and portName variables in the @WebService annotation.

4. Creating and Deploying Web Services

This chapter describes how to create JAX-WS web service applications from Java class and WSDL files, how to deploy and package them, and how to manage them in JEUS 9.

4.1. Overview

The process for deployment is almost the same as for any other web services implemented either from Java class or WSDL file. Since JAX-WS web service implementation is POJO web service-oriented, there is much less number of deployment descriptors than in a JAX-RPC web service.

JAX-WS web services can be programmed without using a deployment descriptor, which was generally used for the implementation of JEUS 5 JAX-RPC web services. This enables a developer to develop web services more conveniently and rapidly (Descriptor free programming).

4.2. Creating and Deploying Java Class Web Services

Creating a web service, implemented from Java classes, as a WAR file involves binding service endpoint implementation classes and other Java classes (portable artifacts) used by the implementation class with multiple deployment descriptor (DD) files into a WAR format.

The following shows the web service implemented from the Java classes from the previous chapter in a WAR format grouped by directory.

```
META-INF/MANIFEST.MF
WEB-INF/classes/AddNumbersImplService_schema1.xsd
WEB-INF/classes/AddNumbersImplService.wsdl
WEB-INF/classes/fromjava/server/AddNumbersImpl.class
WEB-INF/classes/fromjava/server/jaxws/AddNumbers.class
WEB-INF/classes/fromjava/server/jaxws/AddNumbersResponse.class
```

As shown in the previous example, the WAR file consists of a service endpoint implementation class, portable artifact classes, and the web.xml file, which is the only DD file. The web.xml file is not required to implement a JAX-WS web service, but it is used here to obtain a preferred endpoint address. For more details about endpoint addresses, refer to How to Determine an Endpoint Address.

The web service WAR file can be accessed when deployed using the method provided by JEUS 9. To deploy the WAR file in JEUS 9, enter the following command in the console.

```
$ ant build deploy
```

The following is the actual address for accessing the service after a successful deployment.

```
http://host:port/AddNumbers/AddNumbersImplService
```

4.3. Creating and Deploying EJB Web Services

Creating a web service, implemented from a Java class, as a JAR file involves binding a service endpoint implementation class and other Java classes (portable artifacts) used by the implementation class with multiple DD files into a JAR format.

The following shows the web service implemented from the stateless session beans from the previous chapter in a JAR format grouped by directory.

```
META-INF/MANIFEST.MF
fromejb/server/AddNumbersImpl.class
fromejb/server/jaxws/AddNumbers.class
fromejb/server/jaxws/AddNumbersResponse.class
```

As shown in the previous example, the JAR file consists of service endpoint implementation and portable artifact classes. For more information about endpoint addresses, refer to How to Determine an Endpoint Address.

The web service JAR file can be accessed when deployed by using the method provided by JEUS 9. To deploy the JAR file in JEUS 9, enter the following command in the console.

```
$ ant build deploy
```

The following is the actual address for accessing the service after a successful deployment.

```
http://host:port/AddNumbersImplService/AddNumbersImpl
```

4.4. Creating and Deploying WSDL Web Services

Creating a web service, implemented from WSDL, as a WAR file involves binding a service endpoint interface, its service endpoint implementation class, and other portable artifacts with multiple DD files into a WAR file format.

The following shows the web service implemented from the WSDL from the previous chapter in a WAR format grouped by directory.

```
META-INF/MANIFEST.MF
WEB-INF/wsdl/AddNumbers.wsdl
WEB-INF/classes/fromjava/server/AddNumbers.class
WEB-INF/classes/fromjava/server/AddNumbersImpl.class
WEB-INF/classes/fromjava/server/AddNumbersPortType.class
```

```
WEB-INF/classes/fromjava/server/AddNumbersResponse.class
WEB-INF/classes/fromjava/server/AddNumbersService.class
WEB-INF/classes/fromjava/server/ObjectFactory.class
WEB-INF/classes/fromjava/server/package-info.class
```

As shown in the previous example, the WAR file consists of the service endpoint interface (AddNumbersPortType), an implementation of the interface (AddNumbersImpl), multiple portable artifact classes, and a single DD file, web.xml. The web.xml file is not required to implement a JAX-WS web service, but they are used here to obtain a preferred endpoint address. For more information about endpoint addresses, refer to How to Determine an Endpoint Address.

The web service WAR file can be accessed when deployed using the method provided by JEUS 9. To deploy the WAR file in JEUS 9, enter the following command in the console.

```
$ ant build deploy
```

The following is the actual address for accessing the service after a successful deployment.

```
http://host:port/AddNumbers/AddNumbersService
```

4.5. How to Determine an Endpoint Address

In JEUS 9, a JAX-WS web service can be deployed without any deployment descriptor file (descriptor-free), such as web.xml or webservices.xml file, which was required in JEUS 5.

This section discusses how to determine an address to access a JAX-WS web service when the service is deployed to JEUS 9, for servlet based and EJB based web services.

4.5.1. Servlet Endpoint

A servlet endpoint URL is determined based on the following priority order.

1. **When using the <url-pattern> element in web.xml**

   The <url-pattern> element in the web.xml file is used to configure the web service URL. This value overwrites the default value of @EndpointDescription endpoint address.

   The following is an example of the web.xml file that configures the endpoint class named AddNumbersImpl.

   Using the <url-pattern> Element in web.xml : <web.xml>

   ```
   <web-app>
       <display-name>fromwsdl</display-name>
       <description>fromwsdl</description>
       <servlet>
   ```

```
        <servlet-name>fromwsdl</servlet-name>
        <display-name>fromwsdl</display-name>
        <servlet-class>fromwsdl.server.AddNumbersImpl</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>fromwsdl</servlet-name>
<url-pattern>/addnumbers</url-pattern>
    </servlet-mapping>
</web-app>
```

Endpoint Class: <AddNumbersImpl.java>

```
@WebService(serviceName="AddNumbers")
@EndpointDescription(endpointUrl="MyService")
public class AddNumbersImpl {

    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

The following is the actual URL for the web service.

```
http://server:port/context/addnumbers
```

2. **When using the @EndpointDescription Annotation**

   If an endpointUrl variable is specified in the @EndpointDescription annotation, its value is the URL for accessing the web service. The value overwrites the default serviceName endpoint address in the @WebService annotation.

   The following is an example of an endpoint class.

   Using the @EndpointDescription Annotation: <AddNumbersImpl.java>

```
@WebService(serviceName="AddNumbers")
@EndpointDescription(endpointUrl="MyService")
public class AddNumbersImpl {

    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

   The following is the actual URL for the web service.

```
http://server:port/context/MyService
```

3. **When using the serviceName attribute of the @WebService Annotation**

   If a serviceName variable is specified in the @WebService annotation, its value is the URL for the web service. This value overwrites the default endpoint address.

   The following is an example of an endpoint class.

   Using the serviceName Attribute of the @WebService Annotation: <AddNumbersImpl.java>

   ```
   @WebService(serviceName="AddNumbers")
   public class AddNumbersImpl {

       public int addNumbers(int number1, int number2) {
           return number1 + number2;
       }
   }
   ```

   The following is the actual URL for the web service.

   ```
   http://server:port/context/AddNumbers
   ```

4. **Default value of an endpoint address + "Service"**

   If nothing is specified but the @WebService annotation, the default value of the web service endpoint address is the endpoint class name + "Service".

   The following is an example of an endpoint class.

   Endpoint Class: <AddNumbersImpl.java>

   ```
   @WebService
   public class AddNumbersImpl {

       public int addNumbers(int number1, int number2) {
           return number1 + number2;
       }
   }
   ```

   The following is the actual URL for the web service.

   ```
   http://server:port/context/AddNumbersImplService
   ```

4.5.2. EJB Endpoint

The priority order for determining the EJB endpoint URL is the same as that of the servlet endpoint, but the method for determining the context is different.

The context is determined based on the following priority order.

1. **When using the @EndpointDescription Annotation**

   ◦ Endpoint

   If an endpointUrl variable is specified in the @EndpointDescription annotation, its value is the URL for accessing the web service. The value overwrites the default serviceName endpoint address in the @WebService annotation.

   The following is an example of an endpoint class.

   Using the @EndpointDescription Annotation (1) : <AddNumbersImpl.java>

```
@WebService(name="AddNumbersService")
@EndpointDescription(endpointUrl="MyService")
@Stateless
public class AddNumbersImpl {

    public AddNumbersImpl() {

    }

    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

   The following is the actual URL for the web service.

```
http://server:port/context/MyService
```

   ◦ Context

   If a contextPath attribute value is used for the @jeus.webservices.annotation.EndpointDescription annotation, it overwrites the default context value.

   The following is an example of an endpoint class.

   Using the @EndpointDescription Annotation (2) : <AddNumbersImpl.java>

```
@WebService(name="Hello", serviceName="AddNumbersService")
@jeus.webservices.annotation.EndpointDescription(contextPath="EJBService",
    endpointUrl="MyEndpoint")
@Stateless
public class AddNumbersImpl {

    public AddNumbersImpl() {

    }

    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

The following is the actual URL for the web service.

```
http://server:port/EJBService/MyEndpoint
```

2. **When using the serviceName attribute of the @WebService Annotation**
   - Endpoint

     If a serviceName variable is specified in the @WebService annotation, its value is the URL for the web service. This value overwrites the default endpoint address.

     The following is an example of an endpoint class.

     Using the serviceName Attribute of the @WebService Annotation (1) : <AddNumbersImpl.java>

     ```
     @WebService(name="AddNumbersService")
     @Stateless
     public class AddNumbersImpl {

         public AddNumbersImpl() {

         }

         public int addNumbers(int number1, int number2) {
             return number1 + number2;
         }
     }
     ```

     The following is the actual URL for the web service.

     ```
     http://server:port/context/AddNumbersService
     ```

   - Context

     If a serviceName attribute value is used for the @WebService annotation, it overwrites the default context value.

     The following is an example of an endpoint class.

     Using the serviceName Attribute of the @WebService Annotation (2) : <AddNumbersImpl.java>

     ```
     @WebService(name="Hello", serviceName="AddNumbersService")
     @Stateless
     public class AddNumbersImpl {

         public AddNumbersImpl() {

         }
     ```

```
    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

The following is the actual URL for the web service.

```
http://server:port/AddNumbersService/Hello
```

3. **Default value of the context**

   - Endpoint

     If nothing is specified but the @WebService annotation, the default value of the web service endpoint address is the name of the endpoint class.

     The following is an example of an endpoint class.

     Endpoint Class as Default Value of the Context: <AddNumbersImpl.java>

     ```
     @WebService
     @Stateless
     public class AddNumbersImpl {

         public AddNumbersImpl() {

         }

         public int addNumbers(int number1, int number2) {
             return number1 + number2;
         }
     }
     ```

     The following is the actual URL for the web service.

     ```
     http://server:port/context/AddNumbersImpl
     ```

   - Context

     The default value of a context is the service name (ServiceName) from the WSDL document of the web service. The default service name is specified by adding 'Service' to the endpoint class name.

     The following is the address for the web service of this endpoint class.

     ```
     http://server:port/AddNumbersImplService/AddNumbersImpl
     ```

5. Invoking Web Services

This chapter describes how to invoke web services with implementation examples.

5.1. Overview

A JAX-WS web service client application can access a remote web service endpoint using the following approaches:

- Dynamic Proxy
- Dispatch

5.2. Web Service Invocation Using Dynamic Proxy

There are two types of web service invocation facilitated by a dynamic proxy: one in Java SE and the other in Jakarta EE. For both of these categories, client artifacts must be generated in advance.

This section describes the mechanisms of invoking the Java SE and Jakarta EE web services, and how to create the client artifacts.

5.2.1. Creating a Client Artifact

Whether a web service is generated from a Java class or WSDL file, the web service client is invoked by creating a Java class after creating the client artifacts with the published WSDL file. This section briefly shows how to create the client artifacts for a client invoking the web service created by the Java class from the previous chapter.

Web service client artifacts (proxies) consist of service endpoint interface and service interface classes, which were obtained from the WSDL provided by the service by using the JEUS 9 **wsimport** tool.

The following example shows how to obtain the client artifacts by using wsimport in the console.

```
$ >wsimport -help
Usage: wsimport [options] <WSDL_URI>

where [options] include:
  -b <path>                   specify jaxws/jaxb binding files or additional schemas
                              (Each <path> must have its own -b)
  -B<jaxbOption>              Pass this option to JAXB schema compiler
  -catalog <file>             specify catalog file to resolve external
                              entity references supports TR9401,
                              XCatalog, and OASIS XML Catalog format.
  -d <directory>              specify where to place generated output files
  -extension                  allow vendor extensions - functionality
                              not specified by the specification. Use
```

```
                            of extensions may result in applications
                            that are not portable or may not
                            interoperate with other implementations
  -help                     display help
  -httpproxy:<host>:<port>  specify a HTTP proxy server (port defaults to 8080)
  -keep                     keep generated files
  -p <pkg>                  specifies the target package
  -quiet                    suppress wsimport output
  -s <directory>            specify where to place generated source files
  -target <version>         generate code as per the given JAXWS spec version
                            e.g. 2.0 will generate compliant code for JAXWS 2.0 spec
  -verbose                  output messages about what the compiler is doing
  -version                  print version information
  -wsdllocation <location>  @WebServiceClient.wsdlLocation value

Examples:
  wsimport stock.wsdl -b stock.xml -b stock.xjb
  wsimport -d generated http://example.org/stock?wsdl
```

JEUS 9 web services support an ANT TASK for the wsimport tool. For more information about console commands and ANT TASK, refer to "wsimport" console tool and "wsimport" plugin in *JEUS Reference Guide*.

Portable artifacts are created by using wsimport ANT TASK with a WSDL file of the following remote web service. (http://host:port/AddNumbers/AddNumbersImplService?wsdl)

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://server.fromjava/"
    name="AddNumbersImplService" xmlns:tns="http://server.fromjava/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://server.fromjava/"
                schemaLocation="AddNumbersImplService_schema1.xsd" />
        </xsd:schema>
    </types>
    <message name="addNumbers">
        <part name="parameters" element="tns:addNumbers" />
    </message>
    <message name="addNumbersResponse">
        <part name="parameters" element="tns:addNumbersResponse" />
    </message>
    <portType name="AddNumbersImpl">
        <operation name="addNumbers">
            <input message="tns:addNumbers" />
            <output message="tns:addNumbersResponse" />
        </operation>
    </portType>
    <binding name="AddNumbersImplPortBinding" type="tns:AddNumbersImpl">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
            style="document" />
        <operation name="addNumbers">
```

```
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>
    <service name="AddNumbersImplService">
        <port name="AddNumbersImplPort"
            binding="tns:AddNumbersImplPortBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL" />
        </port>
    </service>
</definitions>
```

Creating a Client Artifact: <build.xml>

```
...
<target name="build_client" depends="do-deploy-success, init">
    <antcall target="wsimport">
        <param name="package.name" value="fromjava.client" />
        <param name="binding.file" value="" />
        <param name="wsdl.file"
            value="http://localhost:8088/AddNumbers/AddNumbersImplService?wsdl" />
    </antcall>
    <antcall target="do-compile">
        <param name="javac.excludes" value="fromjava/server/" />
    </antcall>
</target>
...
```

The following are the files that are generated as the result of creating the portable artifacts from the WSDL file of the web service which will be invoked using the aforementioned wsimport ANT TASK.

```
fromwsdl/client/AddNumbers.class
fromwsdl/client/AddNumbersImpl.class
fromwsdl/client/AddNumbersImplService.class
fromwsdl/client/AddNumbersResponse.class
fromwsdl/client/ObjectFactory.class
fromwsdl/client/package-info.class
```

JAXB uses the java bean files, AddNumbers and AddNumbersResponse, to convert a request and response, respectively, for the addNumbers method.

The AddNumbersImpl file implements the service endpoint interface, and the AddNumbersImplService file implements the service interface Java class used as a proxy by the client. The ObjectFactory and packageinfo classes are created by JAXB.

The following is the AddNumbersImplService class which is the service interface class obtained from the WSDL file of the previous remote web service by using wsimport.

Service Interface Class: <AddNumbersImplService.java>

```
@WebServiceClient(name = "AddNumbersImplService",
    targetNamespace = "http://server.fromjava/",
    wsdlLocation = "http://localhost:8088/AddNumbers/AddNumbersImplService?wsdl")
public class AddNumbersImplService extends Service {

    private final static URL ADDNUMBERSIMPLSERVICE_WSDL_LOCATION;
    private final static WebServiceException ADDNUMBERSIMPLSERVICE_EXCEPTION;
    private final static QName ADDNUMBERSIMPLSERVICE_QNAME = new QName("http://server.fromjava/",
"AddNumbersImplService");

    static {
        URL url = null;
        WebServiceException e = null;
        try {
            url = new URL(
                "http://localhost:8088/AddNumbers/AddNumbersImplService?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        ADDNUMBERSIMPLSERVICE_WSDL_LOCATION = url;
        ADDNUMBERSIMPLSERVICE_EXCEPTION = e;
    }

    public AddNumbersImplService() {
        super(__getWsdlLocation(), ADDNUMBERSIMPLSERVICE_QNAME);
    }

    public AddNumbersImplService(WebServiceFeature... features) {
        super(__getWsdlLocation(), ADDNUMBERSIMPLSERVICE_QNAME, features);
    }

    public AddNumbersImplService(URL wsdlLocation) {
        super(wsdlLocation, ADDNUMBERSIMPLSERVICE_QNAME);
    }

    public AddNumbersImplService(URL wsdlLocation, WebServiceFeature... features) {
        super(wsdlLocation, ADDNUMBERSIMPLSERVICE_QNAME, features);
    }

    public AddNumbersImplService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public AddNumbersImplService(URL wsdlLocation, QName serviceName,
                             WebServiceFeature... features) {
        super(wsdlLocation, serviceName, features);
    }

    @WebEndpoint(name = "AddNumbersImplPort")
    public AddNumbersImpl getAddNumbersImplPort() {
     return super.getPort(new QName("http://server.fromjava/", "AddNumbersImplPort"),
                             AddNumbersImpl.class);
    }

    @WebEndpoint(name = "AddNumbersImplPort")
    public AddNumbersImpl getAddNumbersImplPort(WebServiceFeature... features) {
     return super.getPort(new QName("http://server.fromjava/", "AddNumbersImplPort"),
                        AddNumbersImpl.class, features);
    }

    private static URL __getWsdlLocation() {
        if (ADDNUMBERSIMPLSERVICE_EXCEPTION!= null) {
            throw ADDNUMBERSIMPLSERVICE_EXCEPTION;
        }
        return ADDNUMBERSIMPLSERVICE_WSDL_LOCATION;
    }
}
```

The previous service interface is used to obtain the actual proxy object from the client.

The client can obtain the service endpoint interface AddNumbersImpl by using the getAddNumbersImplPort() method of the AddNumbersImplService object, which was instantiated through the AddNumbersImplService constructor of the previous example class.

The following is the AddNumbersImpl class which is a service endpoint interface obtained from the WSDL file of the previous remote web service by using wsimport.

Service Endpoint Interface: <AddNumbersImpl.java>

```
@WebService(name = "AddNumbersImpl", targetNamespace = "http://server.fromjava/")
@XmlSeeAlso( { ObjectFactory.class })
public interface AddNumbersImpl {

    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "addNumbers",
        targetNamespace = "http://server.fromjava/",
        className = "fromjava.client.AddNumbers")
    @ResponseWrapper(localName = "addNumbersResponse",
        targetNamespace = "http://server.fromjava/",
        className = "fromjava.client.AddNumbersResponse")
    public int addNumbers(@WebParam(name = "arg0", targetNamespace = "")
    int arg0, @WebParam(name = "arg1", targetNamespace = "")
    int arg1);
}
```

The service endpoint interface has annotations for converting into Java object, which is used for dynamic binding or at runtime, or XML document. (The service endpoint interface may be used to recreate the WSDL and schema files, but they may not be identical to the WSDL or schema files used to obtain the service endpoint interface.)

The following are Java bean classes, AddNumbers and AddNumbersResponse, used by JAXB to convert a request and response of the addNumbers method into a Java object or XML document.

Java Bean Class: <AddNumbers.java>

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "addNumbers", propOrder = { "arg0", "arg1" })
public class AddNumbers {

    protected int arg0;
    protected int arg1;

    public int getArg0() {
        return arg0;
    }

    public void setArg0(int value) {
        this.arg0 = value;
    }

    public int getArg1() {
        return arg1;
    }

    public void setArg1(int value) {
        this.arg1 = value;
    }
}
```

Java Bean Class: <AddNumbersResponse.java>

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "addNumbersResponse", propOrder = { "_return" })
public class AddNumbersResponse {

    @XmlElement(name = "return")
    protected int _return;

    public int getReturn() {
        return _return;
    }

    public void setReturn(int value) {
        this._return = value;
    }
}
```

The two Java bean classes, AddNumbers and AddNumbersResponse, are message elements in the WSDL file of the remote web service.

5.2.2. Java SE Client Invocation

This section describes how to create and invoke a client class, that contains the logic for invoking a remote web service, by using client portable artifacts obtained in the Creating a Client Artifact section.

5.2.2.1. Creating a Java SE Client Program

Create an AddNumbersImplService object, which is a service interface obtained in the previous example, and then obtain an object which implements the service endpoint interface AddNumbersImpl. The object contains logic for invoking a remote web service through a dynamic proxy. Now call a method to invoke the actual web service by using the object.

The following is a sample code that invokes a remote web service with portable artifacts which were obtained in the previous wsimport ANT TASK execution.

Java SE Client Program: <AddNumbersClient.java>

```
public class AddNumbersClient {

    public static void main(String[] args) {
       AddNumbersImpl port = new AddNumbersImplService().getAddNumbersImplPort();

       int number1 = 10;
       int number2 = 20;

       System.out.println("##############################################");
       System.out.println("### JAX-WS Webservices examples - fromjava ###");
       System.out.println("##############################################");
       System.out.println("Invoking addNumbers(" + number1 + ", " + number2 + ")");
       int result = port.addNumbers(number1, number2);
       System.out.println("Result: " + result);
    }
}
```

5.2.2.2. Invoking a Java SE Client Program

This section discusses how to deploy a web service, which is implemented by using the classes and other configuration files from the previous examples, and execute the client program.

The client program that accesses a remote web service is invoked by using the same method regardless of whether it is implemented from Java or WSDL.

If implemented from Java, create the service in the fromjava directory, and if implemented from WSDL file, create it in the fromwsdl directory. Use the following command to deploy the service to JEUS 9.

```
$ ant build deploy
```

The client must be built after the service has been successfully deployed. Since the Java SE client goes through the wsimport process, the client can only be built once deployment has been completed.

Create the client and invoke the service as in the following.

```
$ ant run
...

run:
     [java] ##################################################
     [java] ##### JAX-WS Webservices examples - fromjava #####
     [java] ##################################################
     [java] Invoking addNumbers(10, 20)
```

```
    [java] Result: 30

...

BUILD SUCCESSFUL
```

The console will show that the client has invoked the service and received the result successfully.

5.2.3. Jakarta EE Client Invocation

This section explains how to use the Jakarta EE method to create and invoke a client that calls a web service generated through the WSDL file from the previous section. It is assumed that the client portable artifacts have already been obtained as stated in the Creating a Client Artifact section.

5.2.3.1. Creating a Jakarta EE Client Program

In generating Jakarta EE clients for a JAX-RPC web service, the <service-ref> element is added to the web.xml file of a servlet or the ejb-jar.xml file of an EJB. This way, the information required to create a client proxy for a web service gets registered with the JNDI. However, the @WebServiceRef annotation needs to be configured to generate Jakarta EE clients for a JAX-WS web service.

The following example shows how to configure the @WebServiceRef Annotation.

```
...
@WebServiceRef(wsdlLocation="http://host:port/TmaxService/TmaxService?wsdl)
static TmaxServiceImplService tsvc;
...
```

Declare a client Java class member variable with the @WebServiceRef annotation by using the AddNumbersService interface. The member variable will automatically inject its value into a servlet container for a servlet or into an EJB container for an EJB, even if there is no set method when the client class is initialized at runtime.

By doing this, AddNumbersPortType, an object which implements the service endpoint interface, can be obtained. The object contains logic for invoking a remote web service through a dynamic proxy. This object can be used to call the method that invokes the actual web service.

The following shows a client program that invokes a remote web service by using portable artifacts obtained through the wsimport ANT TASK execution in the previous example.

Jakarta EE Client: <AddNumbersClient.java>

```
public class AddNumbersClient extends HttpServlet {

    @WebServiceRef
    static AddNumbersImplService svc;

    protected void doGet(HttpServletRequest arg0, HttpServletResponse arg1)
```

```
            throws ServletException, IOException {
        AddNumbersImpl port = svc.getAddNumbersImplPort();

        int number1 = 10;
        int number2 = 20;

        System.out.println("##############################################");
        System.out.println("### JAX-WS Webservices examples - fromjava ###");
        System.out.println("##############################################");
        System.out.println("Invoking addNumbers(" + number1 + ", " + number2 + ")");
        int result = port.addNumbers(number1, number2);
        System.out.println("##############################################");
        System.out.println("### JAX-WS Webservices examples - fromjava ###");
        System.out.println("##############################################");
        System.out.println("Result: " + result);
    }
}
```

5.2.3.2. Invoking a Jakarta EE Client Program

This section discusses how to deploy a web service, which is implemented using the classes and other configuration files from previous examples, and execute the Jakarta EE client program. The client program that accesses a remote web service is invoked by using the same method regardless of whether it is implemented from Java or WSDL.

If implemented from Java, create the service in the fromjava directory, and if implemented from WSDL file, create it in the fromwsdl directory. Use the following command to deploy the service to JEUS 9.

```
$ ant build deploy
```

The client must be built after the service has been successfully deployed. Since the Jakarta EE client goes through the wsimport process, the client can only be built once deployment has been completed.

Create the client and invoke the service as in the following.

```
$ ant run
...

##############################################
### JAX-WS Webservices examples - fromjava ###
##############################################
Invoking addNumbers(10, 20)

...

##############################################
### JAX-WS Webservices examples - fromjava ###
##############################################
Result: 30
```

```
...
```

After entering the previous commands in the console, the web browser can be used to view the server logs to check that the client servlet has invoked the service and received the result successfully.

5.3. Web Service Invocation Using Dispatch

Invoking web services by using the dispatch method is intended for developers who prefer handling the XML configuration at the XML level by using the java.lang.transform.Source or jakarta.xml.soap.SOAPMessage interfaces. Web service invocation using the dispatch method can be done in the message or payload mode, and can be used to create REST web services through XML/HTTP binding (jakarta.activation.DataSource). For more information, refer to Provider and Dispatch Interfaces.

6. Standardized Binding Declaration and Customization

This chapter describes how to declare and customize standardized binding.

6.1. Overview

As stated in the previous chapter, JAX-WS web services can be implemented from Java class or WSDL document.

Various functions can be added to implement more extensible web services. They can be added through wsgen by embedding annotations in Java class files, or through wsimport by adding the functions to WSDL documents.

This section introduces binding customization used to configure a client or web service through the wsimport tool by using WSDL document. The next section will provide the function details, and how to configure and implement a Web service with Java classes by using the wsgen tool.

JEUS Web service supports the binding declaration and customization of Java classes in WSDL file (if mainly using the wsimport tool) through a standardized method as requested by JAX-WS. In the existing JAX-RPC web services, this standard is not specified and web services are not portable across vendors.

The following are the roles that the standardized binding declaration and customization plays in implementing web services.

- Customize mapping from almost all WSDL components to the Java language including service endpoint interface class, method name, parameter name, and exception class.
- Customize functions like asynchronization, provider, wrapper, and additional headers.

6.2. Declaring Standard Binding

Elements related to all bindings belong to the namespace, http://java.sun.com/xml/ns/jaxws , and bindings are declared as the 'jaxws:bindings' element by adding 'jaxws' to the prefix of the namespace.

A binding can be declared in either of the following.

- Declaring in External Documents (Files)
- Declaring in WSDL Documents

6.2.1. Declaring in External Documents (Files)

When declaring binding in an external document (file), the client who is the main user of the web service, passes the WSDL document path as a parameter of the wsimport tool. The path is set in the wsdlLocation element of the binding declaration document.

Declaring in an External Document: <custom-client.xml>

```
<jaxws:bindings
wsdlLocation="http://localhost:8080/AddNumbers/addnumbers?WSDL"
    jaxws:xmlns="http://java.sun.com/xml/ns/jaxws">
...
</jaxws:bindings>
```

To declare component binding in the WSDL document, add it as the child element of the previously declared root element. Place the components inside the node element by using the XPath syntax. Add the binding as a child element of the node element.

The following is an example of adding an element for the binding declaration in the WSDL document.

Declaring Component Binding in a WSDL Document: <custom-client.xml>

```
<jaxws:bindings
    wsdlLocation="http://localhost:8088/AddNumbers/addnumbers?WSDL"
    jaxws:xmlns="http://java.sun.com/xml/ns/jaxws">
<jaxws:bindings node="wsdl:definitions"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
        <jaxws:package name="customize.client"/>
        ...
</jaxws:bindings>
```

As shown in the previous example, the binding declaration means that the package name of Java classes created for wsdl:definitions in the WSDL document will be customized to 'customize.client'.

The created external binding declaration document can be executed using the **wsimport** tool as in the following.

```
$ wsimport -b custom-client.xml http://localhost:8088/AddNumbers/addnumbers?WSDL
```

6.2.2. Declaring in WSDL Documents

The following are the differences in declaring binding directly in the WSDL document instead of in an external document.

- The <jaxws:bindings> element is used as an extension element in the WSDL document.
- The node attribute is not used.
- The <jaxws:bindings> element is not used as a child element of the <jaxws:bindings> element.
- The <jaxws:bindings> element only affects the WSDL component which is wrapping it.

The following is an example of customizing the <wsdl:portType> element in the WSDL document.

Declaring in a WSDL Document: <AddNumbers.wsdl>

```
<wsdl:portType name="AddNumbersImpl">
    <jaxws:bindings xmlns:jaxws="https://jakarta.ee/xml/ns/jaxws"">
<jaxws:class name="MathUtil"/>
        ...
    </jaxws:bindings>
    <wsdl:operation name="addNumber">
    ...
</wsdl:portType>
```

The extension element, <jaxws:bindings>, is the customization of a service endpoint interface class name, which is generated from wsdl:portType. The example shows that the class name, AddNumbersImpl, which is generated with the default value, is customized to 'MathUtil'.

6.3. Standard Binding Customization

This section describes the customization of the declared bindings with the following details.

- Global Bindings
- Customizing a Package Name
- Wrapper Style
- Asynchronization
- Provider Interface
- Customizing Class Name
- Customizing Java Methods
- Customizing Java Parameters
- Customizing XML Schema
- Customizing Handler Chains

6.3.1. Global Bindings

The global bindings are valid from the binding declarations defined in an external document file, and they apply to the entire scope of wsdl:definition in the WSDL document referenced by the jaxws:bindings@wsdlLocation.

These elements have a global scope.

```
<jaxws:package name="..."/>
<jaxws:enableWrapperStyle/>
<jaxws:enableAsyncMapping/>
```

They are used as in the following.

Global Binding: <custom-client.xml>

```
<jaxws:bindings
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="http://localhost:8080/AddNumbers/addnumbers?WSDL"
    xmlns="http://java.sun.com/xml/ns/jaxws">
<package name="customize.client"/>
        <enableWrapperStyle>true</enableWrapperStyle>
        <enableAsyncMapping>false</enableAsyncMapping>
</jaxws:bindings>
```

6.3.2. Customizing a Package Name

When creating a Java class from WSDL document by using the wsimport tool, the class package name is determined by the namespace of the WSDL file.

To customize the class package name, declare the binding customization with the jaxws:package element. The package name can be modified with the '-p' option when created through the wsimport command line, and the modified name overwrites what was previously customized by the jaxws:package element.

Customizing a Package Name (1) : <custom-client.xml>

```
<jaxws:bindings
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="http://localhost:8080/AddNumbers/addnumbers?WSDL"
    xmlns="http://java.sun.com/xml/ns/jaxws">
<package name="customize.client" />
...
</jaxws:bindings>
```

The configuration can be shortened as in the following.

Customizing a Package Name (2) : <custom-client.xml>

```
<jaxws:bindings
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="http://localhost:8080/jaxws-external-customize/addnumbers?WSDL"
    xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions">
<package name="customize.client" />
...
</jaxws:bindings>
```

6.3.3. Wrapper Style

By default, the wsimport tool applies the wrapper style rule to an abstract operation set to the wsdl:portType element in WSDL document. A user can disable the wrapper style through binding customization.

Wrapper Style: <custom-client.xml>

```
<jaxws:bindings
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="http://localhost:8080/AddNumbers/addnumbers?WSDL"
    xmlns="http://java.sun.com/xml/ns/jaxws">
<enableWrapperStyle>true</enableWrapperStyle>
    <bindings node="wsdl:definitions/
        wsdl:portType[@name='AddNumbersImpl']/
        wsdl:operation[@name='addNumbers']">
<enableWrapperStyle>false</enableWrapperStyle>
    ...
</jaxws:bindings>
```

The wrapper style can be set through the jaxws:enableWrapperStyle element, and can be used inside the wsdl:definitions, wsdl:portType and wsdl:operation elements.

- If used inside the wsdl:definitions, it is applied to all wsdl:operations elements of all wsdl:portType attributes.
- If used inside the wsdl:portType, it is applied to all wsdl:operations elements of the portType.
- If used Inside the wsdl:operation element, it is applied only to the operation.

6.3.4. Asynchronization

The client application generates asynchronous polling or callback operations when executing wsimport by declaring the jaxws:enableAsyncMappingbinding element. For more information about asynchronous client applications, refer to Asynchronous Web Services.

The same rules for the wrapper style apply to the components and application scope within the WSDL document.

Asynchronization: <custom-client.xml>

```
<jaxws:bindings
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="http://localhost:8080/AddNumbers/addnumbers?WSDL"
    xmlns="http://java.sun.com/xml/ns/jaxws">
        <enableAsyncMapping>false</enableAsyncMapping>
    <bindings node="wsdl:definitions/
        wsdl:portType[@name='AddNumbersImpl']/
        wsdl:operation[@name='addNumbers']">
        <enableAsyncMapping>true</enableAsyncMapping>
    ...
</jaxws:bindings>
```

6.3.5. Provider Interface

This section describes how to declare a provider interface through binding declaration. For more information, refer to Provider and Dispatch Interfaces.

To declare a port as a provider interface in the WSDL document, use the XPath syntax to declare the wsdl:port node, and then declare the jaxws:provider binding. By default, the provider interface is not declared and is only available when creating a web service from WSDL.

6.3.6. Customizing Class Name

In a WSDL document, wsdl:portType, wsdl:fault, soap:headerfault, and wsdl:server elements are created as Java classes, and the class names can be modified through jaxws:class binding declaration. The following describes each of the Java component classes in the WSDL document.

- Service Endpoint Interface Class

  The following is an example of binding customization for a service endpoint interface class name.

  Service Endpoint Interface Class: <custom-client.xml>

  ```
  <jaxws:bindings node="wsdl:definitions/wsdl:portType[@name='AddNumbersImpl']">
      <jaxws:class name="TmaxUtil" />
  </jaxws:bindings>
  ```

  In the previous example, the service endpoint interface class name is TmaxUtil, instead of the default name, AddNumbersImpl.

- Exception Class

  The following is an example of binding customization for an exception class name.

  Exception Class: <custom-client.xml>

  ```
  <jaxws:bindings node="wsdl:definitions/
      wsdl:portType[@name='AddNumbersImpl']/
      wsdl:operation[@name='addNumbers']/
      wsdl:fault[@name='AddNumbersException']">
      <jaxws:class name="TmaxException" />
  </jaxws:bindings>
  ```

  In this example, the exception class name is TmaxException, instead of the default name, AddNumbersException.

- Service Class

  The following is an example of binding customization for a service class name.

  Service Class: <custom-client.xml>

  ```
  <jaxws:bindings node="wsdl:definitions/
      wsdl:service[@name='AddNumbersService']">
      <jaxws:class name="TmaxService" />
  ```

```
</jaxws:bindings>
```

In the example, the service class name is TmaxService, instead of the default name, AddNumbersService.

6.3.7. Customizing Java Methods

Use the jaxws:method binding declaration to customize a method name for obtaining the service endpoint method name or service class port.

- Service Endpoint Interface Method

  The following is an example of binding declaration for modifying the service endpoint interface method name.

  Service Endpoint Interface Method: <custom-client.xml>

  ```
  <jaxws:bindings node="wsdl:definitions/
      wsdl:portType[@name='AddNumbersImpl']/
      wsdl:operation[@name='addNumbers']">
      <jaxws:method name="add" />
  </jaxws:bindings>
  ```

  The wsimport tool will use 'add', not the default value 'addNumbers', as the method name of the service endpoint interface.

- Service Class Methods to Access the Port

  The following is an example of binding declaration for modifying the name of the service class method for port access.

  Service Class Method to Access the Port : <custom-client.xml>

  ```
  <jaxws:bindings node="wsdl:definitions/
      wsdl:service[@name='AddNumbersService']/
      wsdl:port[@name='AddNumbersImplPort']">
      <jaxws:method name="getTmaxUtil" />
  </jaxws:bindings>
  ```

  The wsimport tool will use 'getTmaxUtil', not the default value 'getAddNumbersImplPort', as the service class method name for port access.

6.3.8. Customizing Java Parameters

The jaxws:parameter binding declaration is used to change the parameter name of the implemented Java method into a preferred one. A user can modify the method parameter name of wsdl:operation or wsdl:portType.

Customizing a Java Parameter: <custom-client.xml>

```
<jaxws:bindings node="wsdl:definitions/
    wsdl:portType[@name='AddNumbersImpl']/
    wsdl:operation[@name='addNumbers']">
    <jaxws:parameter part="definitions/
        message[@name='addNumbers']/
        part[@name='parameters']"
        element="tns:number1" name="num1"/>
</jaxws:bindings>
```

As shown in the previous example, the java parameter customization declaration changes the method parameter name of wsdl:operation, from 'number1' to 'num1'.

6.3.9. Customizing XML Schema

XML schema declared in the WSDL document can be modified by using the standard JAXB binding element inside the jaxws:bindings element.

Customizing an XML Schema (1) : <custom-client.xml>

```
<jaxws:bindings node="wsdl:definitions/
    wsdl:types/
    xsd:schema[@targetNamespace='http://tmaxsoft.com']">
    <jaxb:schemaBindings>
        <jaxb:package name="fromwsdl.server"/>
    </jaxb:schemaBindings>
</jaxws:bindings>
```

In addition, a user can apply binding customization to XML schema files imported by the WSDL document. In such a case, use the JAXB standard extension binding declaration file.

Customizing an XML Schema (2) : <custom-client.xml>

```
<jxb:bindings
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
    version="1.0">
    <jxb:bindings
        schemaLocation=
            "http://localhost:8088/AddNumbers/schema1.xsd"
        node="/xsd:schema">
        <jxb:schemaBindings>
            <jxb:package name="fromjava.client"/>
        </jxb:schemaBindings>
    </jxb:bindings>
...
```

The external standard JAXB binding declaration file is transmitted by using the '-b' option of the wsimport tool.

6.3.10. Customizing Handler Chains

The jaxws:bindings binding declaration is also used to add or customize a handler. For more information about handlers, refer to Handler Framework.

To add or modify handler information to a binding customization declaration, configure the handler chain in jaxws:bindings element, as stated in the schema(JAR 181) for handler chain settings.

Customizing a Handler Chain: <custom-client.xml>

```
<jaxws: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/jaxws-fromwsdlhandler/addnumbers?WSDL"
    xmlns:jaxws="https://jakarta.ee/xml/ns/jaxws"
    xmlns:jakartaee="https://jakarta.ee/xml/ns/jakartaee">
    <jaxws:bindings node="wsdl:definitions">
        <jakartaee:handler-chain>
            <jakartaee:handler-chain-name>
                LoggingHandlers
            </jakartaee:handler-chain-name>
            <jakartaee:handler>
                <jakartaee:handler-name>Logger</jakartaee:handler-name>
                <jakartaee:handler-class>
                    fromwsdlhandler.common.LoggingHandler
                </jakartaee:handler-class>
            </jakartaee:handler>
        </jakartaee:handler-chain>
    </jaxws:bindings>
</jaxws:bindings>
```

When portable artifacts are created by using the wsimport tool with an external binding declaration document or WSDL document, a handler configuration file is also created. When a service endpoint interface class is created, the @jakarta.jws.HandlerChain annotation that is used by JAX-WS runtime to search for handlers is added to the class.

7. Handler Framework

This chapter describes the basic concepts and components of the handler framework with examples.

7.1. Overview

JAX-WS Web services provide a plug-in style framework, which is more convenient for the handlers and also improves the runtime system functionality of JEUS 9 web services.

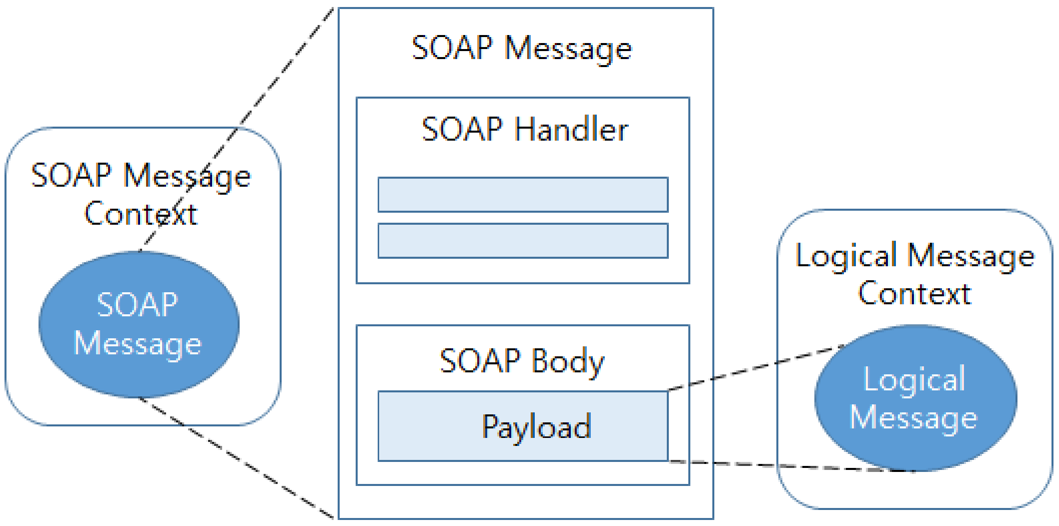
The following are two types of handlers.

- Logical handler

  Accesses the message payload regardless of the protocol.

- SOAP handler

  Accesses the entire content of a SOAP message including the headers.



Relationship between the message contexts

Use the following criteria to select a handler.

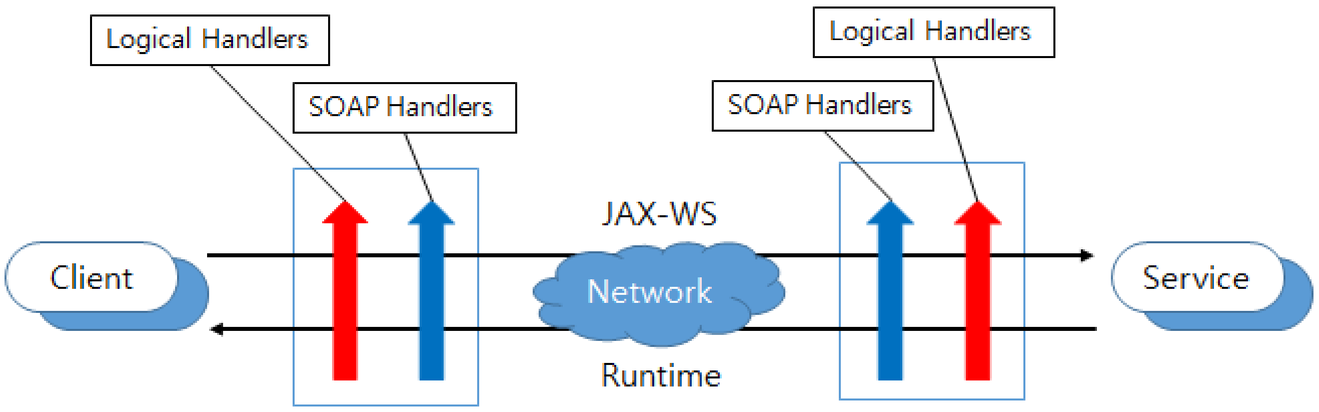
- Use the SOAP handler if the entire SOAP message is needed.
- Use the logical handler if only the XML document payload of the SOAP message is needed.

In other cases, web services configure the endpoint class to process messages. In a special case where the Java object is required, the Interceptor class supported by JEUS EJB is used. For more information about the JEUS EJB Interceptor class, refer to "JEUS EJB Guide".

7.2. Handler Chain Execution Order

For an outbound message, all logical handlers in the handler chain are processed before SOAP handlers. For an inbound message, all SOAP handlers are processed before logical handlers.

In programming client or service-endpoint interfaces, even though logical handlers are configured before SOAP handlers, all logical handlers are processed before SOAP handlers in creating or calling the services.



Handler Framework

7.3. Configuring a Handler Class

This section describes how to organize a handler class and discusses about the MessageContext class.

7.3.1. Declaring Handler Classes

To configure a handler class, a user creates a class that implements a logical handler or SOAP handler interface.

The following is an example of each handler class.

Handler Class: <MyLogicalHandler.java>

```
public class MyLogicalHandler implements
    LogicalHandler<LogicalMessageContext> {
    public boolean handleMessage(LogicalMessageContext messageContext) {
        LogicalMessage msg = messageContext.getMessage();
        return true;
    }
}
```

Handler Class: <MySOAPHandler.java>

```
public class MySOAPHandler implements
    SOAPHandler<SOAPMessageContext> {
    public boolean handleMessage(SOAPMessageContext messageContext) {
        SOAPMessage msg = messageContext.getMessage();
        return true;
    }
}
```

As shown in the previous example, both logical and SOAP handlers implement a handler interface, and the handler interface implements two methods, handlerMessage( ) and handleFault( ).

Each method receives an instance that inherits the MessageContext class as a parameter, and the instance is used to determine whether the message is inbound or outbound. The @PostConstruct and @PreDestroy annotations can be used in a user handler class as shown in the following example.

Handler Class: <MyLogicalHandler.java>

```
public class MyLogicalHandler implements
    LogicalHandler<LogicalMessageContext> {
    @PostConstruct
    public void methodA() {}

    @PreDestroy
    public void methodB() {}
}
```

In the previous MyLogicalHandler class, methodA, declared with the @PostConstruct annotation, is invoked after the handler is created and methodB, declared with the @PreDestroy annotation, is invoked before the handler is destroyed.

7.4. Handler Class Configuration

This section describes how to attach a user-configured handler to a web service.

7.4.1. Creating a Web service from Java Class

To create a web service using a Java class, configure the @HandlerChain annotation in the service endpoint implementation class with the wsgen tool.

Creating a Web service from Java Class: <MyServiceImpl.java>

```
@WebService
@HandlerChain( file="handlers.xml")
public class MyServiceImpl {
    ...
}
```

The following is the handlers.xml file that is set as the @HandlerChain annotation in the previous example. A server handler is configured in the metadata file, handlers.xml.

Creating a Web service from Java Class: <handlers.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<jws:handler-chains xmlns:jws="https://jakarta.ee/xml/ns/jakartaee">
    <jws:handler-chain>
        <jws:handler>
            <jws:handler-class>fromjava.handler.TmaxHandler</jws:handler-class>
        </jws:handler>
    </jws:handler-chain>
</jws:handler-chains>
```

7.4.2. Creating a Web Service from WSDL

To create a web service from WSDL, indirectly embed binding customization into the WSDL document, which is used to create the web service, by using the wsimport tool.

The following is an example of indirectly embedding binding customization into the WSDL document by using an external file.

```
<bindings xmlns="http://java.sun.com/xml/ns/jaxws">
<handler-chains xmlnx="xmlns="https://jakarta.ee/xml/ns/jakartaee">
        <handler-chain>
            <handler>
                <handler-class>fromwsdl.handler.TmaxHandler</handler-class>
            </handler>
        </handler-chain>
    </handler-chains>
</bindings>
```

Note that the <handler-chains> element is added to the binding customization file in this example.

As shown in the following example, multiple <handler-chain> elements can be created inside the <handler-chains> element.

```
<handler-chains xmlnx="xmlns="https://jakarta.ee/xml/ns/jakartaee">
    <handler-chain>
        <service-name-pattern xmlns:tm="http://tmaxsoft.com">
            tm:Tmax*Service
        </service-name-pattern>
        <handler>...</handler>
    </handler-chain>

    <handler-chain>
        <port-name-pattern xmlns:tm="http://tmaxsoft.com">
            tm:TmaxPort
        </port-name-pattern>
        <handler>...</handler>
    </handler-chain>

    <handler-chain>
        <protocol-bindings>##SOAP11_HTTP</protocol-bindings>
        <handler>...</handler>
    </handler-chain>
</handler-chains>
```

If multiple <handler-chain> elements are created inside the <handler-chains> element, some attributes such as service name, port name, or protocol can be applied to any handler.

7.4.3. Creating a Client

A client is created in the same way as creating a web service from WSDL.

7.5. Web Service with a Handler Chain

This section shows a simple example of implementing a user SOAP handler, that is used to output the log, and using it for a web service.

The jakarta.xml.ws.handler.LogicalHandler class, a logical handler class, or the jakarta.xml.ws.handler.SOAPHandler class, a SOAP handler class, inherits the abstract interface, jakarta.xml.ws.handler.Handler. A user can create a handler, as needed, by implementing the two classes.

The aforementioned handler class, LoggingHandler, that will be implemented in the example is a class that implements the SOAP handler class, jakarta.xml.ws.handler.soap.SOAPHandler.

Web Service with a Handler Chain: <LoggingHandler.java>

```
public class LoggingHandler implements SOAPHandler<SOAPMessageContext> {

    public Set<QName> getHeaders() {
        return null;
    }

    public void close(MessageContext messageContext) {

    }

    public boolean handleFault(SOAPMessageContext smc) {
        return true;
    }

    public boolean handleMessage(SOAPMessageContext smc) {
        Boolean inboundProperty =
                (Boolean) smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

        System.out.println("\n##############################################");
        System.out.println("### JAX-WS Webservices examples - handler  ###");
        System.out.println("##############################################");

        if (inboundProperty.booleanValue()) {
            System.out.println("\nClient message:");
        } else {
            System.out.println("\nServer message:");
        }

        SOAPMessage message = smc.getMessage();
        try {
            message.writeTo(System.out);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return true;
    }
}
```

As shown in the previous example, the handler checks whether the message coming into the handler is coming into the server or going out to the client.

Service Class

The following is the service block from the example Java class. In the service block, the @HandlerChain annotation is used to set the handler class of the server.

Web Service Class with a Handler Chain: <handlers.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlnx="xmlns="https://jakarta.ee/xml/ns/jakartaee">
    <handler-chain>
        <handler>
            <handler-class>fromjavahandler.common.LoggingHandler</handler-class>
        </handler>
    </handler-chain>
</handler-chains>
```

The following example shows the handlers.xml file, which is registered as the @HandlerChain annotation.

Web Service Class with a Handler Chain: <AddNumbersImpl.java>

```
@WebService
@HandlerChain(file = "handlers.xml")
public class AddNumbersImpl {

    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

Client Class

The following shows the client block of the example Java class. In the client block, binding user declaration is added when using the wsimport tool to create a client through the WSDL document.

Web Service Client-side Class with a Handler Chain: <AddNumbersClient.java>

```
public class AddNumbersClient {

    public static void main(String[] args) {
        AddNumbersImpl port = new AddNumbersImplService().getAddNumbersImplPort();
        port.addNumbers(10, 20);
    }
}
```

The following example shows the binding declaration.

Binding Declaration in a Web Service with a Handler Chain: <custom-client.xml>

```
<?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:8088/AddNumbers/addnumbers?wsdl"
    xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions"
        xmlns:jws="https://jakarta.ee/xml/ns/jakartaee">
        <jws:handler-chains>
            <jws:handler-chain>
                <jws:handler>
                    <jws:handler-class>
                        fromjavahandler.common.LoggingHandler
                    </jws:handler-class>
                </jws:handler>
            </jws:handler-chain>
        </jws:handler-chains>
    </bindings>
</bindings>
```

7.6. Executing Handler Framework in Web Services

This section describes how to execute the handler framework using the implemented classes and other configuration files from previous examples. Other service endpoint interface implementation classes and configuration files are the same as those from the previous examples.

Create a service by configuring the handler framework and deploy it to JEUS by executing the following command.

```
$ ant build deploy
```

Since the client is processed by using the wsimport tool, it can only be built after the service has been deployed.

As shown in the following example, create a service with the configured handler framework and then call it. In the console, the LoggingHandler outputs the service-client message exchanges on both the service and client screens.

```
$ ant run

...

run:
     [java] ##############################################
     [java] ### JAX-WS Webservices examples - handler  ###
     [java] ##############################################

     [java] Client message:
     [java] <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body><ns2:addNumbers xmlns:ns2="http://server.fromjavahandler/"><arg0>10</arg0>
<arg1>20</arg1></ns2:addNumbers></S:Body></S:Envelope>
     [java] ##############################################
     [java] ### JAX-WS Webservices examples - handler  ###
     [java] ##############################################

     [java] Server message:
     [java] <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Header/><S:Body><ns2:addNumbersResponse xmlns:ns2="http://server.fromjavahandler/">
<return>30</return></ns2:addNumbersResponse></S:Body></S:Envelope>

...

BUILD SUCCESSFUL
```

8. Provider and Dispatch Interfaces

This chapter describes the Provider service endpoint interface and client dispatch interface.

8.1. Overview

A web service and its client communicate with each other through an XML-based messaging. The JAX-WS service endpoint interface provides a high-level abstraction that conceals the complicated conversion processes from a Java object to XML messages and vice versa. However, sometimes a developer may want to process XML message exchanges between a service and client purely at the message level without the help of a complicated programming model such as a SOAP message handler.

JEUS 9 web services implement the **Provider** interface for the service endpoint and a **Dispatch** interface for the client in order to access a message at the message level. The two interfaces enable a web service or client developer to deal with a message at the XML message level.

This chapter describes the two interface types.

8.2. Provider Interface for Service Endpoint

This section describes the Provider interface.

8.2.1. Provider Interface

The Provider interface is used by a web service endpoint to handle an XML document at the XML message level. By doing this, the endpoint can access a message or message payload through a low-level Generic API. 'Generic' is a new function provided by Java SE 5.

The following shows how to use the Provider interface.

1. The @@jakarta.xml.ws.WebServiceProvider annotation must be included.
2. The class must implement the Provider interfaces, javax.xml.transform.Source, jakarta.xml.soap.SOAPMessage, and jakarta.activation.DataSource.
    - Use a message payload by implementing the Provider<javax.xml.transform.Source>.

      To handle a message payload as a source object, set the @ServiceMode annotation to 'Service.Mode.PAYLOAD'. This setting can be omitted if the @WebServiceProvider annotation has been declared as it will be automatically set to the default value.

      ```
      @WebServiceProvider
      public class ProviderImpl implements Provider<Source> {
          public Source invoke(Source source) {
              ...
      ```

```
    }
}
```

- Use a message by implementing the Provider<jakarta.xml.soap.SOAPMessage>

  To handle a message as a SOAPMessage object, set the @ServiceMode annotation to 'Service.Mode.Message'.

  ```
  @WebServiceProvider
  @ServiceMode(value=Service.Mode.MESSAGE)
  public class ProviderImpl implements Provider<SOAPMessage> {
      public SOAPMessage invoke(SOAPMessage msg) {
          ...
      }
  }
  ```

- Use a message by implementing the Provider<jakarta.activation.DataSource>

  To handle a message as a source object, set the @ServiceMode annotation to 'Service.Mode.Message'. If the requested message is a SOAP message, only the SOAPPart excluding the Attachment is transmitted to the Source object. If the return value is null, it means that the web service is one-way.

  ```
  @WebServiceProvider
  @ServiceMode(value=Service.Mode.MESSAGE)
  public class ProviderImpl implements Provider<Source> {
      public Source invoke(Source source) {
          ...
          return null;
      }
  }
  ```

3. The @jakarta.xml.ws.ServiceMode annotation determines whether the endpoint will access a message (Service.Mode.MESSAGE) or the payload (Service.Mode.PAYLOAD).

   If the @jakarta.xml.ws.ServiceMode annotation is not configured, by default it is set to only handle payload.

When creating a web service endpoint interface, since a service implementation class that implements the Provider interface does not provide any information about the endpoint interface, the web service must be created through the WSDL file. When creating a web service through an implementation class that implements the Provider interface, some portable artifact ports obtained from WSDL are not actually used by the implementation class. Thus, an external binding declaration can be used to prevent those ports from being created when the portable artifacts are created through WSDL using the wsimport tool.

8.2.2. Example of a Provider Interface

The following example shows a service implementation class that handles the payload part of a message by implementing the <Provider>Source interface.

Provider Interface: <AddnumbersImpl.java>

```
@ServiceMode(value = Service.Mode.PAYLOAD)
@WebServiceProvider(wsdlLocation = "WEB-INF/wsdl/AddNumbers.wsdl",
    targetNamespace = "http://tmaxsoft.com",
    serviceName = "AddNumbersService", portName = "AddNumbersPort")
public class AddNumbersImpl implements Provider<Source> {
    public Source invoke(Source source) {
        try {
            DOMResult dom = new DOMResult();
            Transformer trans = TransformerFactory.newInstance().newTransformer();
            trans.transform(source, dom);
            Node node = dom.getNode();
            Node root = node.getFirstChild();
            Node first = root.getFirstChild();
            int number1 = Integer.decode(first.getFirstChild().getNodeValue());
            Node second = first.getNextSibling();
            int number2 = Integer.decode(second.getFirstChild().getNodeValue());
            int sum = number1 + number2;

            String body =
            "<ns:addNumbersResponse xmlns:ns=\"http://tmaxsoft.com\"><ns:return>"
            + sum + "</ns:return></ns:addNumbersResponse>";
            Source sumsource =
              new StreamSource(new ByteArrayInputStream(body.getBytes()));

            return sumsource;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

In the previous example, the AddNumbersImpl class implements the <Provider>Source interface through the @WebServiceProvider annotation, and handles the message payload through the @ServiceMode annotation.

8.2.3. Executing a Provider Interface

The following shows how to execute a web service, that implements the Provider interface, by using the implemented classes and other configuration files in this section.

Create a web service by implementing the Provider interface, and deploy it to JEUS by using the following command.

```
$ ant deploy
```

Since the client is processed using the wsimport tool, it can only be built after the service has been deployed.

After creating the client, call the service. The console outputs the details of the message exchanges and the web service that implements the Provider interface.

```
$ ant run

...

run:
     [java] ##############################################
     [java] ### JAX-WS Webservices examples - Provider ###
     [java] ##############################################
     [java] Testing Provider webservices...
     [java] Success!

...

BUILD SUCCESSFUL

...

---[HTTP request]---
Host: localhost:8088
Content-length: 199
Content-type: text/xml; charset=utf-8
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg, *; q=.2,
*/*; q=.2
Connection: keep-alive
Soapaction: ""
User-agent: JAX-WS RI 3.0 - JEUS 9
<?xml version="1.0" ?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envel
ope/"><S:Body><addNumbers xmlns="http://tmaxsoft.com"><arg0>10</arg0><arg1>20</a
rg1></addNumbers></S:Body></S:Envelope>
--------------------
---[HTTP response 200]---
<?xml version="1.0" ?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envel
ope/"><S:Body><ns:addNumbersResponse xmlns:ns="http://tmaxsoft.com"><ns:return>3
0</ns:return></ns:addNumbersResponse></S:Body></S:Envelope>
--------------------

...
```

8.3. Client Dispatch Interface

This section describes the client dispatch interface.

8.3.1. Dispatch Interface

A client application can handle an XML document at the message level, in symmetry with the Provider interface of the service endpoint.

As in the Provide interface for a server, some portable artifact ports obtained from WSDL are not actually used by the service implementation class in a client application that implements the client dispatch interface. Thus, an external binding declaration must be used to prevent those ports from being created when creating the portable artifacts with WSDL by using the wsimport tool.

The following shows how a dispatch interface is used.

1. Implement the javax.xml.transform.Source, jakarta.xml.soap.SOAPMessage, and jakarta.activation.DataSource interfaces.

   The Provider interface of the service endpoint handles a message payload by implementing Provider<Source> and a message by implementing Provider<SOAPMessage> or Provider<Source>. Likewise, a client dispatch interface can form a message by using the SOAPMessage, Source, or JAXB object.

2. Create a service object in order to create a dispatch object (invoke a dynamic service operation).

   The service object (jakarta.xml.ws.Service) plays the role of a factory that creates a dynamic service object. Dynamic service object creation is allocating various binding data of the actual service at runtime in order to create a service object without WSDL.

   The following is an example code used to create a dynamic service.

   ```
   Service service = Service.createService(QName serviceQName);
   Service service = Service.createService(URL wsdlLocation, QName serviceQName);
   ```

   For the created service object, a particular port or endpoint to which a dispatch object is bound to can be added by using the addPort() method, as in the following example. The client application can be invoked through the added port.

   Note that the following methods do not work when the wsdlLocation data is used to dynamically create the service object. This is because the service has already added the port by using the wsdlLocation data.

   ```
   service.addPort(QName portName, String SOAPBinding.SOAP11HTTP_BINDING,
       String endpointAddress);
   service.addPort(QName portName, String SOAPBinding.SOAP12HTTP_BINDING,
       String endpointAddress);
   service.addPort(QName portName, String HTTPBinding.HTTP_BINDING,
       String endpointAddress);
   ```

   The previous example shows how to add a port by using the port name and endpoint address for SOAP 1.1, SOAP 1.2 and XML/HTTP bindings, respectively.

   The aforementioned methods cannot be used with a service object created from a WSDL file. This is because a service object created from portable artifacts already has information about the port binding, port name, or endpoint address, when the service is created from a WSDL file by using the wsimport tool.

The following is an example code for creating a service object from portable artifacts through a general WSDL file by using the wsimport tool.

```
Service service = new AddNumbersService();
```

3. Creating a dispatch object. If the service object was created dynamically or from a WSDL file, a dispatch object is generated by using the two methods of the following service class.

```
Dispatch dispatch = service.createDispatch(QName portName,
    Class clazz, Service.Mode mode);
Dispatch dispatch = service.createDispatch(QName portName,
    JAXBContext jaxbcontext, Service.Mode mode);
```

A dispatch object can use the javax.xml.transform.Source or JAXB data binding object. In both cases, the service object uses the createDispatch method and the Service.Mode.PAYLOAD or Service.Mode.MESSAGE mode to create the dispatch object.

In order for a dispatch object to be able to use the jakarta.xml.soap.SOAPMessage object, the dispatch object must be created from the service object by using the createDispatch method and the Service.Mode.MESSAGE mode.

8.3.2. Example of a Dispatch Interface

The following is an example of a client web service that implements the dispatch interface from a WSDL document of a remote web service.

Dispatch Interface: <AddNumbersClient.java>

```
public class AddNumbersClient {

    public static void main(String[] args) {
        try {
            Service service = new AddNumbersService();
            String request = "<addNumbers xmlns=\"http://tmaxsoft.com\">
                <arg0>10</arg0><arg1>20</arg1></addNumbers>";
            QName portQName = new QName("http://tmaxsoft.com", "AddNumbersPort");
            Dispatch<Source> sourceDispatch = service.createDispatch(portQName,
                    Source.class, Service.Mode.PAYLOAD);
            System.out.println("##############################################");
            System.out.println("### JAX-WS Webservices examples - Dispatch ###");
            System.out.println("##############################################");
            System.out.println("Testing Dispatch webservices...");

            Source result =
              sourceDispatch.invoke(new StreamSource(new StringReader(request)));

            DOMResult dom = new DOMResult();
            Transformer trans = TransformerFactory.newInstance().newTransformer();
            trans.transform(result, dom);
            Node node = dom.getNode();
            Node root = node.getFirstChild();
            Node first = root.getFirstChild();
            int sum = Integer.decode(first.getFirstChild().getNodeValue());
            if (sum == 30) {
                System.out.println("Success!");
            } else {
                System.out.println("Fail!");
            }
        } catch (ProtocolException jex) {
            jex.printStackTrace();
        } catch (TransformerException e) {
            e.printStackTrace();
        }
    }
}
```

As shown in the previous example, the class, AddNumbersClient, obtains a sourceDispatch object of the type Dispatch<Source>, by using the createDispatch() method of the service object created from WSDL.

8.3.3. Executing a Dispatch Interface

The following shows how to execute a web service that implements a dispatch interface by using the implemented classes and other configuration files in this section.

Create the web service and deploy it to JEUS by using the following command.

```
$ ant deploy
```

Once the previous step is completed successfully, build the client. Since the client is processed by using the wsimport tool, it can only be built after the service has been deployed.

As shown in the following example, create a client and invoke the service. The console outputs the details of the message exchanges and the web service that implements the Dispatch interface.

```
$ ant run

...

run:
     [java] ##############################################
     [java] ### JAX-WS Webservices examples - Dispatch ###
     [java] ##############################################
     [java] Testing Dispatch webservices...
     [java] Success!

...

BUILD SUCCESSFUL

...
```

```
---[HTTP request]---
Host: localhost:8088
Content-length: 199
Content-type: text/xml; charset=utf-8
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg, *; q=.2,
*/*; q=.2
Connection: keep-alive
Soapaction: ""
User-agent: JAX-WS RI 3.0 - JEUS 9
<?xml version="1.0" ?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envel
ope/"><S:Body><addNumbers xmlns="http://tmaxsoft.com"><arg0>10</arg0><arg1>20</a
rg1></addNumbers></S:Body></S:Envelope>
--------------------
---[HTTP response 200]---
<?xml version="1.0" ?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envel
ope/"><S:Body><addNumbersResponse xmlns="http://tmaxsoft.com"><return>30</return
></addNumbersResponse></S:Body></S:Envelope>
--------------------

...
```

8.4. XML/HTTP Binding

When exchanging messages in a web service, an XML message is sent over HTTP instead of a SOAP message. Such web service is called a RESTful web service. A service and client of a RESTful web service typically use a provider and dispatch interfaces. For more information about the two interfaces, refer to Provider Interface for Service Endpoint and Client Dispatch Interface.

This section describes XML/HTTP bindings provided by JEUS web service to support RESTful web services.

8.4.1. RESTful Web Services

The following are the main features of a RESTful web service.

- Can be transmitted to the GET Request of HTTP protocol or an endpoint.
- Gets information about the query string and path of the required HTTP request through the standard properties, MessageContext.QUERY_STRING and MessageContext.PATH_INFO.

A RESTful web service does not create a web service and client through a WSDL document. The provider and consumer of a RESTful web service creates the service through a pre-configured schema of the XML document. The provider creates a service class by using the wsgen tool, and the client also creates an application independently without using the portable artifacts.

For more convenience in building RESTful web services, JAX-WS supports XML/HTTP binding in the provider and dispatch interfaces. Through this, the provider and consumer can more easily create a RESTful web service from a JEUS web service.

Creating a Service Endpoint

As explained in Provider Interface for Service Endpoint, in order to implement a provider interface, a service endpoint class that implements Provider<SOAPMessage> or Provider<Source> must be created. The bound Source or SOAPMessage object is used to process the message payload or the entire message.

The provider endpoint can be set to a different binding type by using a binding identifier, which becomes available when the endpoint class includes the @BindingType annotation. If there is no binding identifier, the default binding, SOAP1.1/HTTP, is declared.

The following example shows how to declare an XML/HTTP binding by using the binding identifier.

```
@ServiceMode(value=Service.Mode.MESSAGE)
@BindingType(value=HTTPBinding.HTTP_BINDING)
public class ProviderImpl implements Provider<Source> {
    public Source invoke(Source source) {
        ...
    }
}
```

As shown in the previous example, since the service endpoint class is bound to an inbound message through XML/HTTP, it implements the Provider<Source> interface instead of the Provider<SOAPMessage> interface.

Creating a Client

As described in Dispatch Interface, a client for a RESTful web service is implemented by creating a dynamic service object and a port through 'HTTPBinding.HTTP_BINDING' as in the following example.

```
Service service = Service.createService(QName serviceQName);
service.addPort(QName portName, String HTTPBinding.HTTP_BINDING, String endpointAddress);
```

Create a dispatch object by using the service object from the previous example. If necessary, the service can be invoked by specifying the request as POST or GET in the service object and registering the required query string (MessageContext.QUERY_STRING) and path (MessageContext. PATH_INFO) information.

The following example shows how to create the client-side code.

```
...

Dispatch<Source> d = service.createDispatch(portQName, Source.class, Service.Mode.MESSAGE);
Map<String, Object> requestContext = d.getRequestContext();
requestContext.put(MessageContext.HTTP_REQUEST_METHOD,new String("GET"));
requestContext.put(MessageContext.QUERY_STRING, queryString);
requestContext.put(MessageContext.PATH_INFO, path);
Source result = d.invoke(null);
```

```
...
```

Note that no parameters are passed to the invoke method.

8.4.2. Example of a RESTful Web Service

This section describes examples of a RESTful web service and client.

RESTful Web Service Example

The following example shows a Java class that corresponds to a service of a RESTful web service.

RESTful Web Service: <AddNumbersImpl.java>

```
@WebServiceProvider(serviceName = "AddNumbersService")
@ServiceMode(value = Service.Mode.MESSAGE)
@BindingType(value = HTTPBinding.HTTP_BINDING)
public class AddNumbersImpl implements Provider<Source> {

@Resource(type = Object.class)
    protected WebServiceContext wsContext;

    public Source invoke(Source source) {
        try {
            MessageContext mc = wsContext.getMessageContext();
            String query = (String) mc.get(MessageContext.QUERY_STRING);
            StringTokenizer st = new StringTokenizer(query, "=&/");
            st.nextToken();
            int number1 = Integer.parseInt(st.nextToken());
            st.nextToken();
            int number2 = Integer.parseInt(st.nextToken());
            int sum = number1 + number2;
            String body =
            "<ns:addNumbersResponse xmlns:ns=\"urn:AddNumbers\"><ns:return>"
            + sum + "</ns:return></ns:addNumbersResponse>";
            Source resultsrc =
              new StreamSource(new ByteArrayInputStream(body.getBytes()));
            return resultsrc;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

As shown in the previous example, the RESTful web service declaration sets the BindingType annotation as 'HTTPBinding.HTTP_BINDING', and the Resource annotation as the member variable of the type WebServiceContext, wsContext. JAX-WS RI automatically injects wsContext at runtime.

In summary, the class obtains the MessageContext from WebServiceContext and uses it to get the query string value. Next, it creates a Source object that is parsed and returned by using the sendSource method, and returns the object as a return parameter.

Example of a RESTful Web Service Client

The following is an example of a Java class that corresponds to a client of the RESTful web service.

RESTful Web Service Client: <AddNumbersClient.java>

```
public class AddNumbersClient {

    public static void main(String[] args) throws Exception {
        String endpointAddress = "http://localhost:8088/AddNumbers/addnumbers";
        URL url = new URL(endpointAddress + "?num1=10&num2=20");
        System.out.println("##############################################");
        System.out.println("### JAX-WS Webservices examples - RESTful ###");
        System.out.println("##############################################");
        System.out.println("Testing RESTful webservices...");
        InputStream in = url.openStream();
        StreamSource source = new StreamSource(in);

        try {
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            StreamResult sr = new StreamResult(bos);
            Transformer trans = TransformerFactory.newInstance().newTransformer();
            Properties oprops = new Properties();
            oprops.put(OutputKeys.OMIT_XML_DECLARATION, "yes");
            trans.setOutputProperties(oprops);
            trans.transform(source, sr);
            System.out.println(bos.toString());
            bos.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

As shown in the previous example, in a RESTful web service client, the code for creating portable artifacts from WSDL is not visible. It only shows that the Source object created through a URL object is transmitted in a stream format.

8.4.3. Executing a RESTful Web Service

This section describes how to execute a RESTful web service by using the implemented classes and other configuration files.

Create a RESTful web service and deploy it to JEUS by using the following command.

```
$ ant build deploy
```

Once the service is deployed successfully, build the client.

As shown in the following example, create a client for a RESTful web service and invoke the service. The console outputs the details of the message exchanges between the RESTful web service and client.

```
$ ant run

...

run:
     [java] #############################################
     [java] ### JAX-WS Webservices examples - RESTful ###
     [java] #############################################
     [java] Testing Restful webservices...
     [java] <ns:addNumbersResponse xmlns:ns="urn:AddNumbers"><ns:return>30</ns:r
eturn></ns:addNumbersResponse>

...

BUILD SUCCESSFUL

...

---[HTTP request]---
Host: localhost:8088
Content-type: application/x-www-form-urlencoded
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
User-agent: JAX-WS RI 3.0 - JEUS 9

--------------------
---[HTTP response 200]---
<ns:addNumbersResponse xmlns:ns="urn:AddNumbers"><ns:return>30</ns:return></ns:a
ddNumbersResponse>
--------------------

...
```

9. Asynchronous Web Services

This chapter describes how to configure asynchronous client operations and asynchronous web services that use an asynchronous provider.

9.1. Overview

In invoking a web service between a service and a client, the client blocks the thread until it receives a response from the server. To resolve such performance issues, JAS-WS web service provides asynchronous client operations.

- Asynchronous operation using the client-side JAX-WS API.

  To create a JAX-WS client, use the WSDL file of the service to invoke. A service endpoint interface stub with a static asynchronous method is created through custom binding declarations of the file. A client class that implements the stub is created to implement asynchronous client operations.

- Non-standard asynchronous operation provided by JEUS on the service end.

  This creates a web service endpoint by using an asynchronous web service that operates in an asynchronous method of servlet 3.0. JAX-WS standard does not specify asynchronous operations on the service end.

9.2. Asynchronous Client Operation

This section describes asynchronous operations that use client-side JAX-WS API.

9.2.1. Using a Service Endpoint Interface Stub with Asynchronous Methods

Asynchronous wsdl:operation is mapped to the polling and callback methods. The polling method returns the jakarta.xml.ws.Response interface object, and the callback method returns the jakarta.xml.ws.AsyncHandler interface object. This section describes an asynchronous binding declaration that is used to obtain a service endpoint interface (SEI) stub, which has asynchronous methods, from WSDL.

9.2.1.1. Declaring an Asynchronous Binding

To use the wsdl:operation element for an asynchronized mapping, an SEI based on asynchronous wsdl:operation mapping must be created by using the wsimport tool.

This section introduces how to create such an asynchronous wsdl:operation mapping.

To create asynchronous wsdl:operation mapping, bindings should be customized using the wsimport tool. The following is the binding configuration file, custom-schema.xml.

Asynchronous Binding: <custom-schema.xml>

```
<?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:8088/AddNumbers/addnumbers?wsdl"
    xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions">
        <enableAsyncMapping>true</enableAsyncMapping>
    </bindings>
</bindings>
```

If the binding setting is configured for wsdl:definitions as in the previous example, all wsdl:operation elements in the WSDL document have the same asynchronization setting.

The following is the portion of the build.xml file that uses the custom-schema.xml file.

Asynchronous Binding: <build.xml>

```
...
<target name="build_client" depends="do-deploy-success, init">
    <antcall target="wsimport">
        <param name="package.name" value="async.client" />
        <param name="binding.file" value="-b ${src.conf}/custom-client.xml" />
        <param name="wsdl.file"
            value="http://localhost:8088/AddNumbers/addnumbers?wsdl" />
    </antcall>
    <antcall target="do-compile">
        <param name="javac.excludes" value="fromjava/server/" />
    </antcall>
</target>
...
```

Portable artifacts, which are created through the wsimport tool, are used to create the SEI with asynchronous methods as in the following.

```
public int addNumbers(int number1, int number2)
    throws java.rmi.RemoteException;
public Response<AddNumbersResponse> addNumbers(int number1, int number2);
public Future<?> addNumbers(int number1, int number2,
                           AsyncHandler<AddNumbersResponse>);
```

As shown in the previous example, the two methods (polling and callback), whose return values are Response<AddNumbersResponse> and Future<?>, have been created. The following sub-sections describe how to create a Java client class using the two methods.

9.2.1.2. Configuring an Asynchronous Client

Asynchronous clients can be configured by using the polling or callback method.

How to create a client by using the polling method

The following is the polling method of an asynchronous SEI obtained through the wsimport tool.

```
public Response<AddNumbersResponse> addNumbers(int number1, int number2);
```

The following is an example of a client web service implemented by the method mapped to the polling method. The client application invokes an asynchronous polling method of SEI, and can check for the result.

```
jakarta.xml.ws.Response<AddNumbersResponse> resp = port.addNumbersAsync(10, 20);
while(!resp.isDone()){
}
System.out.println(resp.get().getReturn());
...
```

As shown in the previous example, the method mapped to the polling method returns the jakarta.xml.ws.Response object. The object can determine when the operation is complete by using the isDone() method inherited from java.util.concurrent.Future<T>, and return the result.

The following example shows a part of a client application code that supports the polling method.

Creating a Client with the Polling Method: <AddNumbersClient.java>

```
public class AddNumbersClient {

   ...

   public static void main(String[] args) {
       try {
           AddNumbersImpl port = new AddNumbersService().getAddNumbersImplPort();

           // Asynchronous polling
           Response<AddNumbersResponse> resp = port.addNumbersAsync(10, 20);
           Thread.sleep(2000);
           AddNumbersResponse output = resp.get();
           System.out.println("#############################################");
           System.out.println("### JAX-WS Webservices examples - polling ###");
           System.out.println("#############################################");
           System.out.printf("call webservices in an Asynchronous Polling way...");
           System.out.printf("result : %d\n", output.getReturn());

           ...

       } catch (Exception e) {
           e.printStackTrace();
       }
   }
```

```
    ...

}
```

How to create a client by using the callback method

The following is the callback method of an asynchronous SEI obtained through the wsimport tool.

```
public Future<?> addNumbers(int number1, int number2, AsyncHandler<AddNumbersResponse>);
```

The method mapped to the callback method provides a handler object that implements the jakarta.xml.ws.AsyncHandler interface as an additional parameter.

The following is an example code of a handler object that implements the AsynchHandler interface.

Creating a Client Handler Object with the Callback Method: <AddNumbersClient.java>

```
public class AddNumbersClient {
    ...
    static class AddNumbersCallbackHandler implements
           AsyncHandler<AddNumbersResponse> {

        private AddNumbersResponse output;

        public void handleResponse(Response<AddNumbersResponse> response) {
            try {
                output = response.get();
            } catch (ExecutionException e) {
                e.printStackTrace();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        AddNumbersResponse getResponse() {
            return output;
        }
    }
}
```

The handler object invokes the handleResponse method when it obtains the web service operation result from the server at runtime. The client application can obtain the result by using the getResponse() method.

The following example is a part of client application code that uses the asynchronous callback method of SEI through the handler object. The client application invokes the asynchronous callback method of SEI, and checks for the result.

```
AddNumbersCallbackHandler callbackHandler = new AddNumbersCallbackHandler();
Future<?> resp = port.addNumbersAsync(number1, number2, callbackHandler);
```

```
while(!resp.isDone()){

}
System.out.println(callbackHandler .getResponse().getReturn());
```

As shown in the previous example, the method mapped to the callback method returns the javax.util.concurrent.Future object. The object can determine when the operation is complete by using the isDone() method, and return the result.

The following example is a part of the client application code that uses the asynchronous callback method of SEI.

Creating a Client with the Callback Method: <AddNumbersClient.java>

```
public class AddNumbersClient {

  public static void main(String[] args) {
    try {
        AddNumbersImpl port = new AddNumbersService().getAddNumbersImplPort();
          ...

        // Asynchronous callback
        AddNumbersCallbackHandler callbackHandler = new AddNumbersCallbackHandler();
        Future<?> response = port.addNumbersAsync(10, 20, callbackHandler);
        Thread.sleep(2000);

        output = callbackHandler.getResponse();
        System.out.println("#############################################");
        System.out.println("### JAX-WS Webservices examples - callback ###");
        System.out.println("#############################################");
        System.out.printf("call webservices in an Asynchronous Callback way...");
        System.out.printf("result: %d\n", output.getReturn());
     } catch (Exception e) {
         e.printStackTrace();
     }
  }
    ...
}
```

9.2.1.3. Executing an Asynchronous Client

This section shows how to execute an asynchronous client by using the implemented classes and other configuration files in this section.

Create a web service configured with asynchronous operations and deploy it to JEUS by using the following command.

```
$ ant build deploy
```

Once the service is deployed successfully, build the client.

Create a client configured with the handler framework, and invoke the service from the client.

As shown in the following, the console outputs the result of calling the polling and callback methods that have been processed successfully.

```
$ ant run

...

run:
     [java] #############################################
     [java] ### JAX-WS Webservices examples - polling ###
     [java] #############################################
     [java] call webservices in an Asynchronous Polling way...result : 30
     [java] #############################################
     [java] ### JAX-WS Webservices examples - callback ###
     [java] #############################################
     [java] call webservices in an Asynchronous Callback way...result: 30

...

BUILD SUCCESSFUL
```

9.2.2. Using a Dispatch Interface

Using a client’s asynchronous operation by using a dispatch interface is similar to the description in the previous section. The only difference is that the invokeAsync method, which is provided by the created dispatch object, is used.

The following is an example of using the invokeAsync method.

```
Response<T> response = dispatch.invokeAsync(T);
Future<?> response = dispatch.invokeAsync(T, AsyncHandler);
```

As shown in the previous example, the invokeAsync(T) method supports asynchronization with the polling method, while the invokeAsync(T, AsyncHandler) method uses the callback method. For AsyncHandler, a user handler is implemented as the callback method.

9.3. Asynchronous Web Services

JEUS JAX-WS provides a way to create an asynchronous web service based on servlet 3.0 asynchronous processing. Since a web service endpoint is synchronized during request processing, it occupies the request processing thread allocated by the servlet container increasing the processing time. This can often cause other requests to wait for a thread. In this case, separate threads are assigned for services that require longer processing time, and request processing threads are returned to the container so that the pending requests can be processed efficiently.

This section describes how to configure asynchronous web services.

9.3.1. Configuring Asynchronous Web Services

JEUS provides the following ways to create a JAX-WS web service as an asynchronous web service.

- When using the **@jeus.webservices.jaxws.api.AsyncWebService** annotation

  The following is an example of configuring a web service as an asynchronous web service by using the annotation.

  Configuring an Asynchronous Web Service: <AddNumbersImpl.java>

  ```
  @WebService(serviceName="AddNumbers")
  @AsyncWebService
  public class AddNumbersImpl {

      public int addNumbers(int number1, int number2) {
          return number1 + number2;
      }
  }
  ```

- When using the **the <async-supported> element in web.xml**

  A web service that is already implemented can be configured as an asynchronous web service by using the <async-supported> element, which is supported by servlet 3.0, in web.xml.

  Configuring an Asynchronous Web Service: <web.xml>

  ```
  <web-app>
      <servlet>
          <servlet-name>AddNumbers</servlet-name>
          <servlet-class>fromwsdl.server.AddNumbersImpl</servlet-class>
  <async-supported>true</async-supported>
      </servlet>
      <servlet-mapping>
          <servlet-name>AddNumbers</servlet-name>
          <url-pattern>/addnumbers</url-pattern>
      </servlet-mapping>
  </web-app>
  ```

10. Transmitting Messages Using MIME Attachment

This chapter describes how to transmit a message with a MIME attachment.

10.1. Overview

In implementing a web service, a parameter or return value from a client or server is included in the SOAP message body as a payload, and the message is transmitted over the network. If the parameter or return value is a large binary data containing photo or music, the performance may be compromised.

JAX-WS web services optimize and send such large binary data, defined as the xs:base64Binary or xs:hexBinary element, through MTOM(Message Transmission and Optimization Mechanism) and XOP(XML Binary Optimized Packaging). In addition, it provides a schema called swaRef to transmit all contents of XML elements, which are defined by the schema, as MIME attachments.

In JEUS 9, JAX-WS can send and receive an attachment through the streaming method by using the MessageContext property. This is useful for receiving a large attachment.

10.2. MTOM/XOP

Message Transmission and Optimization Mechanism (MTOM) and XML Binary Optimized Packaging (XOP) define how an XML binary data, such as xs:base64Binary or xs:hexBinary, is optimized for the network and transmitted.

Since an XML type like xs:base64Binary is included in a SOAP envelope when transmitted, the larger the data is the lower the efficiency will be. To resolve this issue, MTOM XOP packages the data into a MIME attachment if the binary data size is larger than the specified value. When the xs:base64Binary or xs:hexBinary element is XOP-packaged into a MIME attachment, the Context-Type of the attachment can be specified as the xmime:expectedContentType attribute value, and the type mapping supported by JAXB is applied to the Context-Type.

An xs:base64Binary or xs:hexBinary schema element is included in a MIME attachment as a MIME type when the size of the element value exceeds the maximum limit. This happens when the element is used with the xmime:expectedContentType attribute setting, which applies the type mapping supported by JAXB to the element. When the xmime:expectedType attribute is not used, it is mapped to a general byte[ ] type. However, if the element value is larger than the maximum value, it is included in the Content-Type of the MIME attachment as a value type like "application/octet-stream".

The following is the JAXB type mapping of the xmime:expectedContentType to a Java type.

MIME Type	Java Type
image/gif	java.awt.Image

MIME Type	Java Type
image/jpeg	java.awt.Image
text/plain	java.lang.String
text/xml or application/xml	javax.xml.transform.Source
/	jakarta.activation.DataHandler

An element such as <element name="image" type="base64Binary"/> is simply mapped to the byte[ ] type, but other elements such as <element name="image" type="base64Binary" xmime:expectedContentTypes="image/jpeg" xmlns:xmime="http://www.w3.org/2005/05/xmlmime"/> are mapped to the java.awt.Image type.

10.2.1. Basic Operations

An element, which is defined as the xs:base64Binary or xs:hexBinary type in wsdl:type of the WSDL document, is defined as the xmime:expectedContentType attribute. When the service interface and portable artifacts are created using a particular type mapping of JAXB through the **wsimport** tool, the client and server can execute the MTOM/XOP environment.

MTOM Operation on a Server

The following example shows how to add the @jakarta.xml.ws.soap.MTOM annotation to a service endpoint implementation class in order to run MTOM on the server side.

```
@jakarta.xml.ws.soap.MTOM
@WebService (endpointInterface = "com.tmax.mtom.Server")
public class ServerImpl implements Server {
    ...
}
```

As an alternative method, MTOM can be run by adding the @BindingType annotation to the service endpoint implementation class as shown in the following example.

```
@BindingType(value=jakarta.xml.ws.SOAPBinding.SOAP11HTTP_MTOM_BINDING)
@BindingType(value=jakarta.xml.ws.SOAPBinding.SOAP12HTTP_MTOM_BINDING)
```

MTOM Operation on a Client

The following example shows how to obtain a proxy or dispatch object through the jakarta.xml.ws.soap.MTOMFeature parameter to run MTOM from the client side.

```
Server port = new ServerService().getServerPort(new MTOMFeature());
jakarta.xml.ws.Service.createDispatch(..., new jakarta.xml.ws.soap.MTOMFeature())
```

The following example shows how to use the obtained proxy or dispatch object to check whether or not MTOM is configured.

```
Server port = new ServerService.getServerPort();
SOAPBinding binding = (SOAPBinding)((BindingProvider)port).getBinding();
boolean mtomEnabled = binding.isMTOMEnabled();
binding.setMTOMEnabled(true);
```

10.2.2. Binary Data Attachment Size Configuration

In a JAX-WS web service, if the size of a java object of the xs:base64Binary or xs:hexBInary type is larger than one kilobyte, it is XOP-encoded as a MIME attachment when it is transmitted from the client or server. The attachment is either packaged into a message or just included in the SOAP message.

When it is XOP-encoded and packaged into a message, the original element is configured with a value of the form <xop:Include href=…>, and the href attribute value of the element is set to the Content-ID of the attachment. The Content-Type of the attachment is set to the xmime:expectedContentTypes attribute value of the type defined as the schema's xs:base64Binary or xs:hexBinary.

The size of the binary data that will be treated as an attachment is set in the following way.

- Server

  The @MTOM annotation is used to configure the server.

  ```
  @jakarta.xml.ws.soap.MTOM(threshold=3000)
  @WebService (endpointInterface = "com.tmax.mtom.Server")
  public class ServerImpl implements Server {
      ...
  }
  ```

- Client

  MTOMFeature class is used to configure the client.

  ```
  Server port = new ServerService().getServerPort(new MTOMFeature(3000));
  jakarta.xml.ws.Service.createDispatch(..., new jakarta.xml.ws.soap.MTOMFeature())
  ```

The following is the actual schema of wsdl:type in WSDL.

```
<element name="Detail" type="types:DetailType"/>
<complexType name="DetailType">
    <sequence>
        <element name="Photo" type="base64Binary"/>
        <element name="image" type="base64Binary"
            xmime:expectedContentTypes="image/jpeg"/>
```

```
    </sequence>
</complexType>
```

The following is the message that is transmitted over the network.

```
Content-Type: Multipart/Related;
start-info="text/xml";
type="application/xop+xml";
boundary="----=_Part_0_1744155.1118953559416"
Content-Length: 3453
SOAPAction: ""

------=_Part_1_4558657.1118953559446

Content-Type: application/xop+xml;
type="text/xml"; charset=utf-8
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Body>
        <Detail xmlns="http://example.org/mtom/data">
            <Photo>RHVrZQ==</Photo>
            <image>
                <xop:Include
                    xmlns:xop="http://www.w3.org/2004/08/xop/include"
                    href="cid:5aeaa450-17f0-4484-b845-a8480c363444@example.org">
                </xop:Include>
            </image>
        </Detail>
    </soapenv:Body>
</soapenv:Envelope>

------=_Part_1_4558657.1118953559446

Content-Type: image/jpeg
Content-ID: <5aeaa450-17f0-4484-b845-a8480c363444@example.org>

╪ α ►JFIF ☺☻ ☺ ☺ █ ♠♠♀¶♀♂♂♀↓↕‼▯¶↔→▼▲↔→∟∟ $.' ",#∟∟(7),

01444▼'9=82<.342 █ C☺ ♀♂♀↑↑2!∟!22222222222222222222222222222222

22222222222 222222 └ ) ¬♥☺" ☻◄☺♥◄☺ — ▼ ☺♣☺☺☺☺☺☺ ☺☻♥♦ ♂

— ╕ ► ☻☺♥♥☻♦♥♣♣♦♦ ☺}☺☻♥ ♦◄♣↕!1A♠‼Qa"q¶2?#B▒┴§R╤≡$3bré ▬
▯↑↓→%&'()*456789:CDEFGHIJSTUVWXYZcdefghijstuvwxyzâäàåçêëèÆôöòûùÿÖÜ
óúñÑªº¿▯¬▒|┤╡╢╖╕╣║╤├─┼╞╟╚╔╩╦╥╙╘╒╓╫╪┘ ┌ßΓπΣσµτΦΘΩ±
≥≤▯▯÷≈°•· —
```

10.2.3. Example of MTOM/XOP

The following is an example of WSDL file to which MTOM/XOP has been applied.

WSDL File Enabling MTOM/XOP: <hello.wsdl>

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:types="http://tmaxsoft.com/mtom/data"
```

```
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://tmaxsoft.com/mtom"
    targetNamespace="http://tmaxsoft.com/mtom" name="mtom">
    <wsdl:types>
        <schema xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://tmaxsoft.com/mtom/data"
            xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
            elementFormDefault="qualified">
            <complexType name="DetailType">
                <sequence>
<element name="image" type="base64Binary"
                        xmime:expectedContentTypes="image/jpeg" />
                </sequence>
            </complexType>
            <element name="Detail" type="types:DetailType" />
            <element name="DetailResponse" type="types:DetailType" />
        </schema>
    </wsdl:types>
    <wsdl:message name="HelloIn">
        <wsdl:part name="data" element="types:Detail" />
    </wsdl:message>
    <wsdl:message name="HelloOut">
        <wsdl:part name="data" element="types:DetailResponse" />
    </wsdl:message>
    <wsdl:portType name="Hello">
        <wsdl:operation name="Detail">
            <wsdl:input message="tns:HelloIn" />
            <wsdl:output message="tns:HelloOut" />
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloBinding" type="tns:Hello">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="Detail">
            <soap:operation />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HelloService">
        <wsdl:port name="HelloPort" binding="tns:HelloBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

As shown in the previous example, the base64Binary type of the image element is defined in the schema of the DetailType element, and it can be managed by MTOM. Since the image type attribute is declared as xmime:expectedContentTypes="image/ jpeg", the Content-Type will be set to "image/jpeg" when it is handled as an attachment by MTOM.

10.2.4. Executing MTOM/XOP Example

This section describes how to execute a handler framework by using the implemented classes and other configuration files in this section.

Create a service configured to use MTOM, and deploy it to JEUS.

```
$ ant deploy
```

Once the previous step is completed successfully, build the client. Since the client is processed by using the wsimport tool, it can only be built after the service has been deployed.

As shown in the following example, create a client in which MTOM is configured and invoke the service. Enter the command in the console, and the console will display the message exchanges between the client and service.

```
$ ant run

...

Host: localhost:8088
Content-length: 4848
Content-type: multipart/related;type="application/xop+xml";boundary="uuid:23ba19
3c-bd1a-4323-abdd-339bf5c05cd1";start-info="text/xml"
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg, *; q=.2,
*/*; q=.2
Connection: keep-alive
Soapaction:
User-agent: JAX-WS RI 3.0 - JEUS 9
--uuid:23ba193c-bd1a-4323-abdd-339bf5c05cd1
Content-Type: application/xop+xml;charset=utf-8;type="text/xml"
Content-Transfer-Encoding: binary

<?xml version="1.0" ?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envel
ope/"><S:Body><Detail xmlns="http://tmaxsoft.com/mtom/data"><image><Include xmln
s="http://www.w3.org/2004/08/xop/include" href="cid:dc0fa852-3d46-4bac-9ad4-889d
59c6198a@example.jaxws.sun.com"/></image></Detail></S:Body></S:Envelope>
--uuid:23ba193c-bd1a-4323-abdd-339bf5c05cd1
Content-Type: image/jpeg
Content-Id: <dc0fa852-3d46-4bac-9ad4-889d59c6198a@example.jaxws.sun.com>
Content-Transfer-Encoding: binary

??JFIF

...
```

10.3. swaRef

When a JAX-WS web service is created by defining an element in wsdl:type of WSDL as wsi:swaRef, a schema type, the transmitted message includes the element value as a MIME attachment.

The element value in the SOAP message body is a reference to the attachment. The element of wsi:swaRef schema is mapped to a Java class of the jakarta.activation.DataHandler type.

10.3.1. Using swaRef

The wsi:swaRef type of XML element is mapped to the DataHandler java class, but it is actually transmitted as an attachment over the network.

For example, the XML schema is defined in WSDL as in the following.

```
<element name="claimForm" type="wsi:swaRef"
    xmlns:wsi="http://ws-i.org/profiles/basic/1.1/xsd"/>
```

If a user creates a web service with the WSDL document by using the wsimport tool and transmits a message over the network, the message will look like the following.

```
Content-Type: Multipart/Related;
start-info="text/xml";
type="application/xop+xml";
boundary="----=_Part_4_32542424.1118953563492"
Content-Length: 1193
SOAPAction: ""

------=_Part_5_32550604.1118953563502

Content-Type: application/xop+xml;
type="text/xml"; charset=utf-8
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<claimForm xmlns="http://example.org/mtom/data">
cid:b0a597fd-5ef7-4f0c-9d85-6666239f1d25@example.jaxws.sun.com
</claimForm>
 </soapenv:Body>
</soapenv:Envelope>

------=_Part_5_32550604.1118953563502

Content-Type: application/xml
Content-ID:
    <b0a597fd-5ef7-4f0c-9d85-6666239f1d25@example.jaxws.sun.com>
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocaption= "https://jakarta.ee/xml/ns/jakartaee
         https://jakarta.ee/xml/ns/jakartaee/application_9.xsd"
         version="9">
    <display-name>Simple example of application</display-name>
    <description>Simple example</description>
    <module>
        <ejb>ejb1.jar</ejb>
    </module>
    <module>
```

```
        <ejb>ejb2.jar</ejb>
    </module>
    <module>
        <web>
            <web-uri>web.war</web-uri>
            <context-root>web</context-root>
        </web>
    </module>
</application>
```

10.3.2. Example of swaRef

The following is an example of a WSDL file to which swaRef has been applied.

WSDL File Enabling swaRef: <hello.wsdl>

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:types="http://tmaxsoft.com/swaref/data"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://tmaxsoft.com/swaref"
    targetNamespace="http://tmaxsoft.com/swaref" name="swaref">
    <wsdl:types>
        <schema xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://tmaxsoft.com/swaref/data"
            xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
            elementFormDefault="qualified"
            xmlns:ref="http://ws-i.org/profiles/basic/1.1/xsd">
            <import namespace="http://ws-i.org/profiles/basic/1.1/xsd"
                schemaLocation="wsi-swa.xsd" />
<element name="claimForm" type="ref:swaRef" />
            <element name="claimFormResponse" type="ref:swaRef" />
        </schema>
    </wsdl:types>
    <wsdl:message name="claimFormIn">
        <wsdl:part name="data" element="types:claimForm" />
    </wsdl:message>
    <wsdl:message name="claimFormOut">
        <wsdl:part name="data" element="types:claimFormResponse" />
    </wsdl:message>
    <wsdl:portType name="Hello">
        <wsdl:operation name="claimForm">
            <wsdl:input message="tns:claimFormIn" />
            <wsdl:output message="tns:claimFormOut" />
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloBinding" type="tns:Hello">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="claimForm">
            <soap:operation />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
```

```
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HelloService">
        <wsdl:port name="HelloPort" binding="tns:HelloBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

As shown in the previous example, the claimForm and claimFormResponse elements are defined as ref:swaRef, a swaRef data type, by using the external schema, wsi-swa.xsd. Hence, they will be transmitted as an attachment over the network.

10.3.3. Executing swaRef Example

This section describes how to execute a handler framework by using the implemented classes and other configuration files in this section.

Create a service configured to use swaRef, and deploy it to JEUS.

```
$ ant build deploy
```

Once the previous step is completed successfully, build the client. Since the client is processed by using the wsimport tool, it can only be built after the service has been deployed.

As shown in the following example, create a client in which swaRef is configured and invoke the service. Enter the command in the console, and the console will display the message exchanges between the client and service.

```
$ ant run
...

Host: localhost:8088
Content-length: 2672
Content-type: multipart/related; type="text/xml"; boundary="uuid:26973efe-2e29-4
36a-8fce-3fa58b6fd91f"
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg, *; q=.2,
*/*; q=.2
Connection: keep-alive
Soapaction:
User-agent: JAX-WS RI 3.0 - JEUS 9
--uuid:26973efe-2e29-436a-8fce-3fa58b6fd91f
Content-Type: text/xml

<?xml version="1.0" ?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envel
ope/"><S:Body><claimForm xmlns="http://tmaxsoft.com/swaref/data">cid:40b15647-3c
0b-4449-90ad-29b1667e940b@example.jaxws.sun.com</claimForm></S:Body></S:Envelope
>
--uuid:26973efe-2e29-436a-8fce-3fa58b6fd91f
```

```
Content-Id:<40b15647-3c0b-4449-90ad-29b1667e940b@example.jaxws.sun.com>
Content-Type: text/xml
Content-Transfer-Encoding: binary

<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:types="http://tmaxsoft.com/swaref/data"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://tmaxsoft.com/swaref"
    targetNamespace="http://tmaxsoft.com/swaref" name="swaref">
...
```

10.4. Streaming Attachments

When transmitting and receiving a SOAP message with an attachment, JEUS 9 JAX-WS web service can send or receive the attachment by using the Streaming method instead of being processed in memory. This function improves the performance of invoking a web service endpoint when the attached file size is large. Especially when the file size is larger than the JVM Heap size, this function can be used to send and receive the attachment without an OutOfMemory error.

This section describes how a web service client transmits an attachment included in the Outbound SOAP message by using the streaming method.

Set a web service client to transmit an outbound request message with chunked transfer-encoding. Set the following property for the requested MessageContext from the service port, before invoking a web service operation on the service port (sending an outbound request message).

- Property Key : "com.sun.xml.ws.transport.http.client.streaming.chunk.size"
- Property Value : chunked data size

This property enables JAX-WS HTTP transport to transmit an HTTP request by using the Chunked Streaming method, which is supported by JEUS 9 server.

Streaming Attachments: <AttachmentApp.java>

```
Hello port = new HelloService().getHelloPort(
    new jeus.webservices.jaxws.api.transport.http.AttachmentFeature());

Map<String, Object> reqCnt = ((BindingProvider)port).getRequestContext();
reqCnt.put("com.sun.xml.ws.transport.http.client.streaming.chunk.size",
            new Integer(4*1024));
```

Note that this method is not supported by all HTTP servers.

11. Fast Infoset Web Services

This chapter introduces a new JEUS web service type that uses Fast Infoset, and describes how to implement and use the service.

11.1. Overview

The Fast Infoset standard specifies effective XML infosets, which can be used in place of XML, in a binary format.

The Fast Infoset specification with the full name "ITU-T Rec. X.891 | ISO/IEC 24824-1 Fast Infoset" is defined by the following organizations.

- ITU-T (International Telecommunications Union - Telecommunication Standardization Sector

  ```
  http://www.itu.int/ITU-T/studygroups/com17/index.asp
  ```

- ISO(International Standards Organization)

  ```
  http://www.iso.org/iso/home/standards_development/list_of_iso_technical_committees.htm
  ```

Like an XML document, a **Fast Infoset document** denotes a binary instance of such XML Infosets. An XML document and Fast Infoset document internally contain XML Infoset. The Fast Infoset document can be serialized or parsed faster and the size is smaller than that of an XML document. A Fast Infoset document can be used when the processing speed and size of an XML document are a concern.

Designing Fast Infoset

A Fast Infoset document can be serialized or parsed like an XML document.

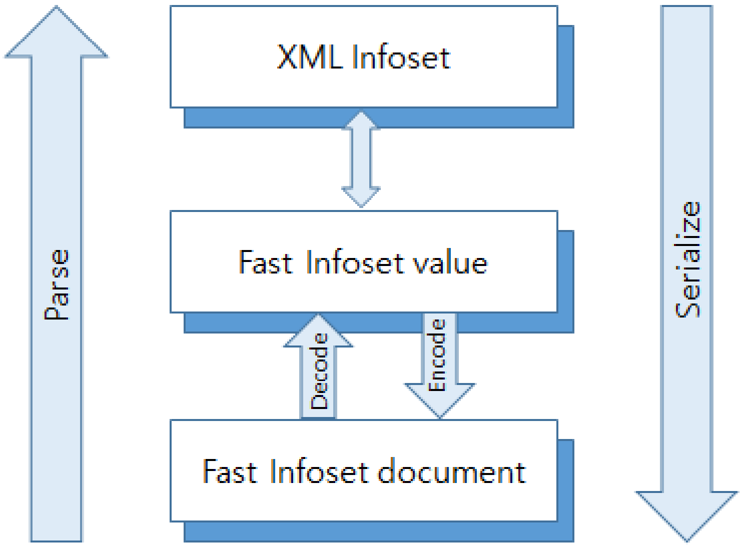
- Serialization process

  Serialization process involves obtaining Fast Infoset from XML Infoset that corresponds to an SAX event or a DOM document obtained from an XML document. The Fast Infoset is used to obtain a Fast Infoset document.

- Parsing process

  Parsing process involves obtaining a Fast Infoset from the Fast Infoset document, and then obtaining an XML Infoset from the Fast Infoset. The Fast Infoset specification guarantees an effective serialization and parsing processes as well as optimal file size reduction.

The following figure shows the serialization and parsing processes.



Serializing/Parsing Fast Infoset

Advantages of Fast Infoset

A Fast Infoset document can be serialized and parsed faster and can be compressed more than an XML document.

Other advantages are:

- No end-tag like the one used in an XML document exists.
- No escape character data exists.
- The decoder that converts a Fast Infoset document into Fast Infoset already has the length information.
- Indexing of repeated strings.
- Indexing of namespaces.
- Can contain binary contents.
- Preserves the states of documents with similar vocabularies.

Advantages of binary Fast Infoset documents are:

- Element items or character information are encoded into smaller bits.
- Fast Infoset encoder and decoder can be implemented easily and efficiently due to well-defined boundaries.
- Speed can be enhanced when serialized or parsed by using indexing information due to the reduced target size.
- Suitable for using the streaming function.

11.2. Using Fast Infoset

This section describes the ContentNegotiation, a mechanism for using the Fast Infoset function.

11.2.1. Content Negotiation

To use the Fast Infoset function in a JAX-WS web service, the user must specify whether to use the function for Accept, which is the standard HTTP header, or Content-Type. Since the client, who invokes the web service, always decides whether to use Fast Infoset and implements it through the HTTP header, this technique is referred as **Content Negotiation**.

The first SOAP message transmitted from the Web service's Fast Infoset, which is decided by whether the client inserts information into the HTTP header, is XML- encoded. If the client includes information about MIME type 'application/fastinfoset' in the HTTP Accept header, all SOAP messages, after the first transmission, are encoded by using Fast Infoset as long as the server supports Fast Infoset. In other words, if the client transmits an XML encoded SOAP message to MIME type 'application/fastinfoset' in the HTTP Accept header, the server, that supports Fast Infoset, transmits a response message encoded by Fast Infoset. If the client later transmits another message through the same stub object, the client and server will exchange messages encoded by Fast Infoset. Such technique is referred as **Negotiation Pessimistic**.

Use the following configuration to use Fast Infoset in the JAX-WS web service.

- Configure client property.

  ```
  ((BindingProvider) stubOrDispatch).getRequestContext().put(
  com.sun.xml.ws.client.ContentNegotiation.PROPERTY, "pessimistic");
  ```

- Configure a system variable of the client VM.

  ```
  java -Dcom.sun.xml.ws.client.ContentNegotiation=pessimistic
  ```

11.3. Example of Fast Infoset

To use Fast Infoset in a JAX-WS web service, a user can configure the client property in the client application, or the system variable on the client VM.

This section describes the former.

Functions provided through other options are not available on the server. As already mentioned, all Fast Infoset properties are configured by the client.

The following is an example of configuring the Fast Infoset property in a client application.

Fast Infoset Example: <AddNumbersClient.java>

```
public class AddNumbersClient {
    public static void main(String[] args) {
      AddNumbersImpl port = new AddNumbersImplService().getAddNumbersImplPort();
      ((BindingProvider) port).getRequestContext().put(
```

```
                ContentNegotiation.PROPERTY, "pessimistic");

        int number1 = 10;
        int number2 = 20;

        System.out.println("##################################################");
        System.out.println("### JAX-WS Webservices examples - fastinfoset ###");
        System.out.println("##################################################");
        System.out.println("Testing Fast Infoset webservices...");
        int result = port.addNumbers(number1, number2);
        if (result == 30) {
            System.out.println("Success!");
        }
    }
}
```

As shown in the previous example, set the 'pessimistic' property to ContentNegotiation of the request context through a proxy(stub) object obtained from the service interface.

11.4. Executing Fast Infoset Web Services

The following shows how to execute a Fast Infoset Web service by using the implemented classes and other configuration files in this section. Other SEI implementation classes and configuration files are the same as those in the examples of the previous section.

Create a service with Fast Infoset Web service configured, and deploy it to JEUS.

```
$ ant build deploy
```

Since the client is processed by using the wsimport tool, it can only be built after the service has been deployed.

As shown in the following example, create a client in which Fast Infoset is configured and then invoke the service.

```
$ ant run

...

run:
     [java] ##################################################
     [java] ### JAX-WS Webservices examples - fastinfoset ###
     [java] ##################################################
     [java] Testing Fast Infoset webservices...
     [java] Success!

...

BUILD SUCCESSFUL
```

```
$ ant run


---[HTTP request]---
Host: localhost:8088
Content-length: 218
Content-type: text/xml; charset=utf-8
Accept: application/fastinfoset, text/xml, multipart/related, text/html, image/g
if, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
Soapaction: ""
User-agent: JAX-WS RI 3.0 - JEUS 9
<?xml version="1.0" ?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envel
ope/"><S:Body><ns2:addNumbers xmlns:ns2="http://server.fastinfoset/"><arg0>10</a
rg0><arg1>20</arg1></ns2:addNumbers></S:Body></S:Envelope>
--------------------
---[HTTP response 200]---
?
```

12. JAX-WS JMS Transport

This chapter describes JAX-WS JMS transport in Web services.

12.1. Overview

In general, JEUS JAX-WS can invoke JEUS web services by using JMS transport, while web service client applications use HTTP to invoke web services.

JAX-WS JMS transport uses the ConnectionFactory and Destination settings of the JEUS MQ server. The web service that uses JAX-WS JMS transport adds the @jeus.webservices.jaxws.api.JMSWebService annotation to the web service endpoint.

If a web service that is using JAX-WS JMS transport is deployed, two wsdl:port elements are defined in the distributed WSDL, an HTTP port and JMS port. The web service client application can choose any type of port to invoke the web service.

12.2. Configuring JAX-WS JMS Transport

This section describes how to configure JAX-WS JMS transport with the assumption that the developer is familiar with JAX-WS web services and JMS.

12.2.1. Configuring a JMS Server

JAX-WS JMS transport works on a JMS server by using the ConnectionFactory and Destination properties of the server. JAX-WS JMS transport uses a queue type as a Destination. A <connection-factory> and <destination> properties must be configured in the **domain.xml** file on the JEUS domain server. For detailed information about the configuration, refer to "JEUS MQ Guide".

The following is an example of configuring JAX-WS JMS transport.

Configuring a JMS Server: <domain.xml>

```
<jms-engine xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    ...
    <connection-factory>
        <type>queue</type>
        <name>QueueConnectionFactory</name>
        <service>jmstest</service>
    </connection-factory>
    ...
</jms-engine>
<jms-resource>
    ...
    <destination>
        <type>queue</type>
        <name>ExamplesQueue</name>
        <multiple-receiver>true</multiple-receiver>
    </destination>
    ...
</jms-resource>
```

12.2.2. Creating a Web Service

To use JMS transport instead of HTTP, add the @jeus.webservices.jaxws.api.JMSWebService annotation to the web service endpoint.

The following is a web service endpoint that includes the @JMSWebService annotation.

Web Service Endpoint: <AddNumbersImpl.java>

```
@WebService
@jeus.webservices.jaxws.api.JMSWebService(
    jndiConnectionFactoryName = "QueueConnectionFactory",
    destinationName = "ExamplesQueue",
    targetService = "AddNumbersImplService")
public class AddNumbersImpl {
...
```

The following are the JMS WebService properties.

Property	Description
jndiConnectionFactoryName	ConnectionFactory name of the JMS Server. It should be identical to the value, domain.xml#<connection-factory>/< name>.
destinationName	Destination name of the JMS Server. It should be identical to the value, domain.xml#<destination>/<name>.
targetService	Service endpoint implementation that will send the message arrived at the destination. If the property is not specified in the fromJava model, the wsdl:service name is used as the targetService property value. Must be identical to the JMS URI targetService property value in the fromWSDL model.

12.2.3. Configuring a WSDL

To create a web service endpoint from WSDL, a wsdl:port for JMS transport and a wsdl:port that uses HTTP must be configured in the WSDL of the deployed application (WAR or JAR).

The following is an example of specifying JMS URI in wsdl:port's soap:address for JAX-WS JMS transport in WSDL.

Configuring a WSDL: <AddNumbers.wsdl>

```
<definitions name="AddNumbers" targetNamespace="urn:AddNumbers" ... >
    ...
    <service name="AddNumbersService">
    <port name="AddNumbersPort" binding="impl:AddNumbersBinding">
        <soap:address location="http://<host>:<port>/<context>/AddNumbersService" />
    </port>
    <port name="AddNumbersJMSPort" binding="impl:AddNumbersBinding">
        <soap:address
location="jms:jndi:ExamplesQueue?targetService=AddNumbersImplService&jndiConnectionFactoryName=QueueC
onnectionFactory" />
    </port>
    </service>
</definitions>
```

SOAP/JMS' JMS URI must be specified according to the 'SOAP over Java Message Service 1.0' standard.

JMS URI uses the "jms" scheme, and specifies the destination name to query JMS destination with JNDI. JMS URI’s targetService property must be identical to the web service endpoint’s @JMSWebService#targetService property value.

12.2.4. Creating a Web Service Client

A web service client that uses JAX-WS JMS transport can call a web service through JMS by using a JMS port instead of the existing HTTP port.

The published WSDL of the web service has the wsdl:port information that is used for JMS transport. A service class that extends jakarta.xml.ws.Service created from the WSDL through the wsimport tool has a method for obtaining a JMS port, and one for obtaining the existing HTTP port.

As shown in the following example, a web service client application can invoke a web service by selecting a port type.

Web Service Client: <AddNumbersClient.java>

```
//AddNumbersPortType port = new AddNumbersService().getAddNumbersPort();
AddNumbersPortType port = new AddNumbersService().getAddNumbersJMSPort();
...
```

13. Web Service Policy

This chapter describes the basic concepts and a simple scenario for configuring and managing the web service policy.

13.1. Overview

JEUS web services support the web service policy (WS-Policy). The web service policy is a standard specification that allows web service functions, such as WS-addressing, WS-RM, WSTX, and WS-security, to expose their policies.

The web service policy that JEUS web services support is divided into the server and client policies. A server can expose its policies through WSDL. When creating a web service client through the exposed policies, the client is automatically configured with the functions specified by the policies.

For more information about the web service policy setting, refer to Web Services Addressing, Reliable Messaging, Web Service Transactions, and Web Service Security.

13.2. Web Service Policy (WS-Policy)

This section introduces the general web service policy (WS-Policy).

The following are the features of web service policies.

- The web service policy specification is designed to be extensible and flexible in expression.
- The web service policy is expressed through one or more policy assertion(s).

For more information about the schema of the standard web service policy, refer to http://schemas.xmlsoap.org/ws/2004/09/policy/ws-policy.xsd.

- Web Service Policy Framework*

The following describes the web service policy framework.

- Policy Container

  A key component of the web service policy framework is the policy container expressed as the 'Policy' element. The element can be referred to or reused by others through the assigned ID. In addition, the element consists of an assertion or a group of assertions. The assertions are made up of policy operators.

- Policy Operator

  The web service policy specification defines two operators and one attribute.

◦ ExactlyOne Operator

  This operator is used to select one assertion or operator as a policy, when there are multiple assertions or operators in a child element.

  The following is an example of using the operator.

```
<wsp:Policy>
    <wsp:ExactlyOne>
        <wsse:SecurityToken>
            <wsse:Token
        ...
        <wsse:
    ...
</wsp:Policy>
```

◦ All Operator

  This operator is used to set a collection of all assertions or operators in the child element as a policy.

  The following is an example of using the operator.

```
<wsp:Policy>
    <wsp:All>
        <wsse:SecurityToken>
            <wsse:Token
    ...
</wsp:Policy>
```

◦ Optional Operator

  This operator is used to selectively set some assertions or operators as a policy, when they are declared as the attributes of a child element.

  The following is an example of using the operator.

```
<wsp:Policy>
    <wsse:Integrity wsp:optional="true">
    ...
</wsp:Policy>
```

13.3. Server Policy

This section describes two scenarios for setting the web service policy, a scenario of creating a web service from WSDL and one for creating a web service from Java classes.

13.3.1. Creating a Web Service from WSDL

The following scenario shows how to create a web service configured with the web service policy setting by using a WSDL document.

1. Create a WSDL document.
2. Set the web service policy in the WSDL document.
3. Create Java bean objects by using the wsimport tool.
4. Create the service implementation classes.
5. Deploy the packaged service to JEUS server.

Directory Structure

By using a WSDL document, create a web service configured with the web service policy setting. The following is the directory that is on the server where a web service is created from a WSDL document.

```
war_root
    |- WEB-INF
        |- classes
        |   |- ... (SEI, JAX-WS artifacts, Handler, Validator)
        |- wsdl
            |- addnumbers.wsdl
```

13.3.2. Creating a Web Service from Java Class

To create a web service with the web service policy setting from Java classes, the wsit-endpoint.xml file must be generated by using the '-policy' option for the **wsgen** tool, as shown in the following example.

```
$ wsgen fromjava.server.AddNumbersImpl -d web/WEB-INF -policy service-config.xml
```

The following is the service-config.xml file. This section will only cover the bolded parts of the following example.

Creating a Web Service with Java Classes: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
        <!-- To apply to the whole of the endpoint -->
        <endpoint-policy-subject>
            <addressing-policy>
                <using-addressing>true</using-addressing>
            </addressing-policy>
            <!-- To apply to an operation(method) of the endpoint -->
            <operation-policy-subject>
                <!-- This setting applies to the below operation (method). -->
                <operation-java-name>addNumbers</operation-java-name>
                <!-- To apply to a request message from the client -->
                <input-message-policy-subject>
                    ...
                </input-message-policy-subject>
                <!-- To apply to a response message from the server -->
                <output-message-policy-subject>
                    ...
                </output-message-policy-subject>
            </operation-policy-subject>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

The following scenario shows how to create a web service configured with the web service policy setting when the wsit-endpoint.xml file and Java classes for the web service are obtained by using wsgen with the '-policy' option.

1. Create the service implementation classes.
2. Create the service-config.xml file by using the 'jeus-webservices-config.xsd' schema.
3. Create the wsit-endpoint.xml file by using wsgen with the '-policy' option when creating a web service by using the created service implementation classes.
4. Save the wsit-endpoint.xml file in the 'WEB-INF' directory that will be packaged.
5. Deploy the packaged service to JEUS server.

Directory Structure

By using Java classes, create a web service configured with the web service policy setting. The following is the directory that is on the server where a web service is created from Java classes.

```
war_root
    |- WEB-INF
          |- classes
          |    |- ... (SEI, JAX-WS artifacts, Handler, Validator)
          |- wsit-Endpoint.xml
```

13.4. Client Policy

In general, web service security scenario is used for the web service policy setting on clients. This is because a JEUS web service is exposed to the web service policy setting in WSDL of a remote web service at runtime, and automatically provides an appropriate environment for the policy. However, there may be some cases, such as in the web service security, where additional settings may be required.

The following is a scenario for creating a client when additional settings are required for the web service with the web service policy setting.

1. Create client Java bean objects through the wsimport tool.
2. Save a remote WSDL document in an accessible repository with the name 'wsit-client.xml'.

The web service policy setting in the remote WSDL can be deleted, since JEUS web service provides a client environment through the web service policy setting at runtime.

3. Set an additional web service policy for the client in 'wsit-client.xml'.
4. For JAR packaging, save 'wsit-client.xml' in the 'classes/META-INF' directory that will be packaged.

   For WAR packaging, put 'wsit-client.xml' under the 'WEB-INF' directory that will be packaged.
5. Deploy the packaged service to JEUS server.

Directory Structure

In general, a web service client executed on a container looks like the following example.

```
war_root
    |- WEB-INF
    |      |- classes
    |            |- ... (client classes, JAX-WS artifacts, Handler, Validator)
    |            |- META-INF
    |                 |- wsit-client.xml
    |- index.jsp
```

A web service client executed on an EJB container or as an independent application looks like the following example.

```
jar_root
    |- classes
         |- ... (client classes, JAX-WS artifacts, Handler, Validator)
         |- META-INF
               |- wsit-client.xml
```

14. Web Services Addressing

This chapter describes transport-independent web services addressing with some examples.

14.1. Overview

Web services addressing is used to define an address for a web service message in a transport-independent way. Since a web service is basically exchanging of stateless messages, web services addressing enables the support for stateful web services. Web services addressing is a basic technology included in many WS-* specifications, including WS-RM, WS-Security, WS-secure Conversation, and WS-Trust, which will be discussed in the following chapters.

Web services addressing works with either of the transport methods, HTTP or SMTP. The following shows how web services addressing works at the message level. If a user tries to add information to a message from the application level without setting the web services addressing configuration, the message will look like the following example.

```
POST /AddNumbers/addnumbers HTTP 1.1/POST
Host: tmaxsoft.com
SOAPAction: http://tmaxsoft.com/AddNumbers/addnumbers

<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
            xmlns:tmax="http://tmaxsoft.com/">
    <S:Header>
        <tmax:MessageID>
            uuid:e197db59-0982-4c9c-9702-4234d204f7f4
        </tmax:MessageID>
    </S:Header>
    <S:Body>
        ...
    </S:Body>
</S:Envelope>
```

As shown in the previous example, the message ID given by the application is used only at the application level, and cannot be used in other applications. In addition, if the transport method is converted from HTTP to SMTP, this creates more difficulties, such as the need to convert the information in the transport header using the mapping rule. To resolve these difficulties, W3C specifies the binding to a SOAP message or WSDL document by defining message addressing properties (MAP) for web services addressing.

The following shows a web services addressing enabled message that includes the MAP information at the message level.

```
POST /AddNumbers/addnumbers HTTP 1.1/POST
Host: tmaxsoft.com
SOAPAction: http://tmaxsoft.com/AddNumbers/addnumbers

<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
            xmlns:wsa="http://www.w3.org/2005/08/addressing/">
```

```
    <S:Header>
        <wsa:MessageID>
            uuid:e197db59-0982-4c9c-9702-4234d204f7f4
        </wsa:MessageID>
        <wsa:To>
            http://tmaxsoft.com/AddNumbers/addnumbers
        </wsa:To>
        <wsa:Action>
            http://tmaxsoft.com/AddNumbers/addnumbers
        </wsa:Action>
    </S:Header>
    <S:Body>
        ...
    </S:Body>
</S:Envelope>
```

A web services addressing enabled SOAP message contains web services addressing related elements such as <wsa:MessageID>, <wsa:To>, and <wsa:Action>. They are bound to a SOAP message that contains the MAP information.

The following describes each element.

Element	Description
<wsa:MessageID>	Absolute URI that uniquely identifies the message.
<wsa:To>	Absolute URI that shows the receiver’s address.
<wsa:Action>	Absolute URI that shows what the message means.

Such a web services addressing enabled message has a specific format that enables the processing of all contents in the message in a transport-independent and application-independent way. For instance, to transmit such a message through a transport method, besides HTTP, like SMTP, the only thing that needs to be done is to modify the wsa:To header value to a value like "mailto:purchasing@example.com".

This section describes how to configure web services addressing and execution method with examples.

14.2. Server Configurations

Web services addressing can be configured from either a Java class or WSDL. This section describes each method.

14.2.1. Configuring from Java Class

To configure web services addressing from a Java class, simply add the jakarta.xml.ws.soap.Addressing and jakarta.jws.WebService annotations.

Configuring Server-Side Web Service Addressing (1): <AddnumbersImpl.java>

```
@Addressing
@WebService
public class AddNumbersImpl {
    ...
}
```

The following steps are performed by the service endpoint when web services addressing is configured through the jakarta.xml.ws.soap.Addressing annotation as shown in the previous example.

- Creates the standard <wsaw:UsingAddressing> element inside the <wsdl:binding> element of the created WSDL document.
- Interprets and examines all web services addressing headers of the message for correct grammar.
- Transmits an error message if the grammar is incorrect.
- Transmits an error message if the <wsa:Action> header value is not identical to the expected value for the operation.
- All transmitted messages contain web services addressing header related information.

Through web services addressing, a user can set an Action MAP (Message Addressing Property) for a service endpoint interface method (the operation element in WSDL document) as in the following example.

Configuring Server-Side Web Service Addressing (2): <AddnumbersImpl.java>

```
@Addressing
@WebService
public class AddNumbersImpl {

    @Action(
        input = "http://tmaxsoft.com/input",
        output = "http://tmaxsoft.com/output",
        fault = {
            @FaultAction(className = AddNumbersException.class,
            value = "http://tmaxsoft.com/fault")
        }
    )
public int addNumbers(int number1, int number2)
        throws AddNumbersException {

        ...

    }
}
```

When converted to WSDL, the method is bound to the operation element in the following WSDL document.

Configuring Server-Side Web Service Addressing (3): <Addnumbers.wsdl>

```
...

<operation name="addNumbers">
    <input wsaw:Action="http://tmaxsoft.com/input"
           message="tns:addNumbers"/>
    <output wsaw:Action="http://tmaxsoft.com/output"
            message="tns:addNumbersResponse"/>
    <fault wsaw:Action="http://tmaxsoft.com/fault"
           message="tns:AddNumbersException"
           name="AddNumbersException"/>
</operation>

...

<binding name="AddNumbersImplPortBinding" type="tns:AddNumbersImpl">
<wsaw:UsingAddressing />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document" />
    <operation name="addNumbers">
        <soap:operation soapAction="" />
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />
        </output>
    </operation>
    <operation name="addNumbers2">
        <soap:operation soapAction="" />
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />
        </output>
    </operation>
</binding>
...
```

14.2.2. Configuring from WSDL

To create a web service from WSDL, use the standard extension element, <wsaw:UsingAddressing>. By using the wsimport tool, the web service can create another web service that supports addressing.

14.3. Client Configurations

The **wsimport** tool automatically determines the addressing setting of the client web service depending on whether the <wsaw:UsingAddressing> element is included in the WSDL document, which is referenced when creating a client.

Sometimes, the web services addressing function may not be executed when creating a client from the service if the client is separately executing the function at the application level.

The following code disables web services addressing from running on the client.

```
new AddNumbersImplService().getAddNumbersImplPort(
    new jakarta.xml.ws.AddressingFeature(false));
```

As shown in the previous example, if jakarta.xml.ws.AddressingFeature is set to false when obtaining a proxy (endpoint interface) from a client, the client can transmit a message without running the web services addressing defined in WSDL.

When WSDL of the service does not include 'wsaw:UsingAddressing' that enables addressing, portable artifacts obtained through the **wsimport** tool can only perform web service functions. To enable web services addressing separately on the client, use methods supported by the jakarta.xml.ws.Service class, or use the jakarta.xml.ws.soap.AddressingFeature class.

```
<T> Dispatch<T> createDispatch(javax.xml.namespace.QName,
    java.lang.Class<T>, Service.Mode, WebServiceFeature...)
Dispatch<java.lang.Object> createDispatch(javax.xml.namespace.QName,
    jakarta.xml.bind.JAXBContext, Service.Mode, WebServiceFeature...)
<T> T getPort(java.lang.Class<T>, WebServiceFeature...)
<T> T getPort(javax.xml.namespace.QName, java.lang.Class<T>,
    WebServiceFeature...)

new AddNumbersImplService().getAddNumbersImplPort(
    new jakarta.xml.ws.AddressingFeature());
```

14.4. Example

The following is an example of the service endpoint implementation class.

Web Service Addressing: <AddNumbersImpl.java>

```
@Addressing
@WebService
public class AddNumbersImpl {

@Resource
    WebServiceContext wsc;

@Action(input = "http://tmaxsoft.com/input",
        output = "http://tmaxsoft.com/output")

    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }

    public int addNumbers2(int number1, int number2) {
        return number1 + number2;
    }
}
```

As shown in the previous example, web services addressing is configured by using the 'action' annotation, which will be bound to the 'addressing' annotation and the 'operation' element in the WSDL document.

14.5. Executing the Example

The following example shows how to create a web service by using addressing enabled endpoint classes and other configuration files, and deploy it to JEUS 9.

```
$ ant deploy
```

Once the service is successfully deployed, execute the client.

The client and server consoles will display the result that the message has been successfully transmitted.

```
$ ant run

...

run:
     [java] ################################################
     [java] ### JAX-WS Webservices examples - addressing ###
     [java] ################################################
     [java] basic name mapping result: 20
     [java] default name mapping result: 20

BUILD SUCCESSFUL

...
...

---[HTTP request]---
Host: localhost:8088
Content-length: 626
Content-type: text/xml; charset=utf-8
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg,
*; q=.2, */*; q=.2
Connection: keep-alive
Soapaction: "http://tmaxsoft.com/input"
User-agent: JAX-WS RI 3.0 - JEUS 9
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <S:Header>
        <wsa:To>http://localhost:8088/AddNumbers/addnumbers</wsa:To>
        <wsa:Action>http://tmaxsoft.com/input</wsa:Action>
        <wsa:ReplyTo xmlns:wsa="http://www.w3.org/2005/08/addressing">
            <wsa:Address>
```

```
                http://www.w3.org/2005/08/addressing/anonymous
            </wsa:Address>
        </wsa:ReplyTo>
        <wsa:MessageID>uuid:880f3891-d07b-4ad1-bbc1-8dce8f1aedef</wsa:MessageID>
    </S:Header>
    <S:Body>
        <ns2:addNumbers xmlns:ns2="http://server.wsaddressing/">
            <arg0>10</arg0><arg1>10</arg1>
        </ns2:addNumbers>
    </S:Body>
</S:Envelope>
---[HTTP response 200]---
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <S:Header>
        <wsa:To>http://www.w3.org/2005/08/addressing/anonymous</wsa:To>
        <wsa:Action>http://tmaxsoft.com/output</wsa:Action>
        <wsa:MessageID>uuid:815cb296-a2f3-45df-80c5-5a2c1ca836ca</wsa:MessageID>
        <wsa:RelatesTo>uuid:880f3891-d07b-4ad1-bbc1-8dce8f1aedef</wsa:RelatesTo>
    </S:Header>
    <S:Body>
        <ns2:addNumbersResponse xmlns:ns2="http://server.wsaddressing/">
            <return>20</return>
        </ns2:addNumbersResponse>
    </S:Body>
</S:Envelope>
--------------------

...

---[HTTP request]---
Host: localhost:8088
Content-length: 663
Content-type: text/xml; charset=utf-8
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg,
*; q=.2, */*; q=.2
Connection: keep-alive
Soapaction: "http://server.wsaddressing/AddNumbersImpl/addNumbers2Request"
User-agent: JAX-WS RI 3.0 - JEUS 9
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <S:Header>
        <wsa:To>http://localhost:8088/AddNumbers/addnumbers</wsa:To>
        <wsa:Action>
            http://server.wsaddressing/AddNumbersImpl/addNumbers2Request
        </wsa:Action>
        <wsa:ReplyTo xmlns:wsa="http://www.w3.org/2005/08/addressing">
            <wsa:Address>
                http://www.w3.org/2005/08/addressing/anonymous
            </wsa:Address>
        </wsa:ReplyTo>
        <wsa:MessageID>uuid:bf65c920-9129-495a-b9dc-8cb8efb9c2a6</wsa:MessageID>
    </S:Header>
    <S:Body>
        <ns2:addNumbers2 xmlns:ns2="http://server.wsaddressing/">
            <arg0>10</arg0><arg1>10</arg1>
        </ns2:addNumbers2>
    </S:Body>
</S:Envelope>
---[HTTP response 200]---
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <S:Header>
        <wsa:To>http://www.w3.org/2005/08/addressing/anonymous</wsa:To>
        <wsa:Action>
            http://server.wsaddressing/AddNumbersImpl/addNumbers2Response
        </wsa:Action>
        <wsa:MessageID>uuid:41975002-46e8-4219-89a6-ed6e72899fa8</wsa:MessageID>
        <wsa:RelatesTo>uuid:bf65c920-9129-495a-b9dc-8cb8efb9c2a6</wsa:RelatesTo>
    </S:Header>
    <S:Body>
        <ns2:addNumbers2Response xmlns:ns2="http://server.wsaddressing/">
            <return>20</return>
        </ns2:addNumbers2Response>
    </S:Body>
</S:Envelope>
--------------------
...
```

15. Reliable Messaging

This chapter describes the concept of reliable messaging and how to configure it.

15.1. Overview

Reliable messaging is intended for creating more reliable web services with high quality of service (QoS). Reliability is measured by the ability to transmit a message from one place to another. The main purpose of reliable messaging is to guarantee application message transmission from/to each end of a web service.

Reliable messaging technology guarantees that messages are transmitted to both ends of the web service exactly once and in order upon request. It is used to recover a message that failed to be transmitted. If a message gets lost during transmission, the sender attempts to re-transmit the message until it receives an acknowledgement from the recipient. If the message arrives in the wrong order, the system that receives the message modifies the order and transmits the message to the web service endpoint in the correct order.

The following are the specifications for reliable messaging.

- WS-Reliable Messaging
- WS-Coordination
- WS-AtomicTransactions

Reconsider using reliable messaging if the following problems occur.

- If a communication failure causes an unavailable network or a connection drop
- If an application message gets lost during transmission
- If application messages are not transmitted in the correct order when the ordered delivery option is used

Also consider the following strengths and weaknesses of reliable messaging.

- Reliable messaging guarantees that a message is transmitted from a source to a destination exactly once. If the ordered delivery of the messages has been set, the messages will be transmitted in order.
- Transmitting a web service message through reliable messaging may degrade the overall performance of the web service. Using the ordered delivery option will degrade performance.
- A client that does not use reliable messaging is not interoperable with a web service that uses it.

15.2. Server Configurations

WS-reliable messaging can be configured on a server by using WSDL or Java classes.

15.2.1. Configuring from WSDL

To implement WS-Reliable Messaging in WSDL, configure the web service policy in the WSDL document and use it to create a web service through the **wsimport** tool.

The following are the steps for configuring WS-Reliable Messaging in the WSDL file according to the web service policy setting.

- Set the following value to PolicyAssertion.

```
<wsrm:RMAssertion xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy" />
```

- Set the following value to PolicyAssertion to guarantee ordered delivery.

```
<rmp:Ordered xmlns:rmp="http://sun.com/2006/03/rm" />
```

The following is the WSDL file with the previous settings.

Configuring Server-Side WS-Reliable Messaging: <AddNumbers.wsdl>

```
<?xml version="1.0" ?>
<definitions name="AddNumbers" targetNamespace="http://example.org"
    xmlns:tns="http://example.org" xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
<wsp:UsingPolicy />
    <wsp:Policy wsu:Id="AddNumbers_policy">
        <wsp:ExactlyOne>
            <wsp:All>
                <wsaw:UsingAddressing />
                <wsrm:RMAssertion>
                    <wsrm:InactivityTimeout Milliseconds="600" />
                    <wsrm:AcknowledgementInterval Milliseconds="200" />
                </wsrm:RMAssertion>
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>
<types>
        <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            targetNamespace="http://example.org">
            <element name="addNumbersResponse" type="tns:addNumbersResponse" />
            <complexType name="addNumbersResponse">
                <sequence>
                    <element name="return" type="xsd:int" />
                </sequence>
            </complexType>
            <element name="addNumbers" type="tns:addNumbers" />
            <complexType name="addNumbers">
```

```
                <sequence>
                    <element name="arg0" type="xsd:int" />
                    <element name="arg1" type="xsd:int" />
                </sequence>
            </complexType>
        </xsd:schema>
    </types>
    <message name="addNumbers">
        <part name="parameters" element="tns:addNumbers" />
    </message>
    <message name="addNumbersResponse">
        <part name="result" element="tns:addNumbersResponse" />
    </message>
    <portType name="AddNumbersPortType">
        <operation name="addNumbers">
            <input message="tns:addNumbers" name="add" />
            <output message="tns:addNumbersResponse" name="addResponse" />
        </operation>
    </portType>
    <binding name="AddNumbersBinding" type="tns:AddNumbersPortType">
<wsp:PolicyReference URI="#AddNumbers_policy" />
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
            style="document" />
        <operation name="addNumbers">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>
    <service name="AddNumbersService">
        <port name="AddNumbersPort" binding="tns:AddNumbersBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL" />
        </port>
    </service>
</definitions>
```

15.2.2. Configuring from Java Class

To configure WS-Reliable Messaging in Java classes, use the '-policy' option of the **wsgen** console tool to obtain the wsit-endpoint.xml file.

```
$ wsgen fromjava.server.AddNumbersImpl -d web/WEB-INF -policy service-config.xml
```

The following is the service-config.xml file.

Configuring WS-Reliable Messaging in a Java Class: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
```

```
        <endpoint-policy-subject>
            <addressing-policy/>
            <rm-policy>
                <inactivityTimeout>600000</inactivityTimeout>
                <acknowledgementInterval>1000</acknowledgementInterval>
            </rm-policy>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

15.3. Client Configurations

There are no additional settings for WS-Reliable Messaging on the client side. JEUS web services automatically provide a suitable environment for WS-Reliable Messaging by interpreting the WS-Reliable Messaging policy in WSDL of a remote web service at client runtime.

15.4. Example

The implementation in a Java class is the same as that of JEUS 9 web service. The only difference is that the deployment descriptor file, 'wsit-endpoint.xml', is added to the 'WEB-INF' folder as a WAR or EAR package.

The following is an example of the service-config.xml file.

Configuring WS-Reliable Messaging: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
        <endpoint-policy-subject>
            <addressing-policy/>
            <rm-policy>
                <inactivityTimeout>600000</inactivityTimeout>
                <acknowledgementInterval>1000</acknowledgementInterval>
            </rm-policy>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

15.5. Executing the Example

Execute the following command to create a web service by using wsgen with the previous service-config.xml file.

```
$ ant build
```

When the web service is created successfully, the structure will look like the following.

```
web
 |
 +-- WEB-INF
        |
        +-- wsit-fromjava.server.AddNumbersImpl.xml
        +-- classes
               |
               +-- fromjava.server.AddNumbersImpl
```

The wsit-fromjava.server.AddNumbersImpl.xml file is saved in the 'WEB-INF' folder as a WAR or EAR package.

To deploy the service, execute the following command.

```
$ ant deploy
```

After the web service is deployed successfully, create a client and invoke the service as in the following example.

```
$ ant run

...

run:
     [java] ##########################################
     [java] ### JAX-WS Webservices examples - wsit ###
     [java] ##########################################
     [java] Testing wsit webservices...
     [java] Success!

...

BUILD SUCCESSFUL
```

16. Web Service Transactions

This chapter describes the concept of a web service transaction and how to configure it.

16.1. Overview

Transaction is a fundamental concept behind the implementation of reliable distributed applications. This mechanism ensures that all transaction participants receive the same result.

In essence, a transaction is characterized by the ACID properties.

Type	Description
Atomicity	All or nothing. All updates complete successfully or they all fail.
Consistency	Only valid data are applied to the database.
Isolated	Transaction result must not affect other concurrent transactions until the transaction has been committed.
Durability	After the transaction has been committed, the updates can be recovered even in case of a system failure.

To control the operation and result of an application, a web service environment needs to provide the same coordinated behavior that is provided by the traditional transaction mechanism. It also requires a functionality that coordinates the processing of results from multiple services which requires a more flexible type of transaction.

The following are the transaction specifications supported by JEUS 9 web services.

- WS-Coordination
- WS-AtomicTransaction

These transaction specifications provide a WSDL definition for such coordinated behavior.

16.2. Server Configurations

A web service transaction for a server application is defined in a WSDL or Java class.

16.2.1. Configuring from WSDL

To create a web service transaction in WSDL, define a web service policy in the WSDL document and use it to create a web service through the wsimport tool.

To define a web service transaction in a WSDL file according to the web service policy, set the PolicyAssertion property as in the following.

```
<wsat200410:ATAssertion xmlns:ns1="http://schemas.xmlsoap.org/ws/2002/12/policy" />
```

The following WSDL file defines a web service transaction.

Configuring a Web Service Transaction in WSDL: <AddNumbers.wsdl>

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="AddNumbersService" targetNamespace="http://server.fromwsdl/"
    xmlns:tns="http://server.fromwsdl/" xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
    oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<wsp:Policy xmlns:wsat200410="http://schemas.xmlsoap.org/ws/2004/10/wsat"
        wsu:Id="AddNumbersPortBinding_addNumbers_WSAT_Policy" wsp:Name="">
        <wsat200410:ATAssertion
            xmlns:ns1="http://schemas.xmlsoap.org/ws/2002/12/policy"
            wsp:Optional="true" ns1:Optional="true" />
    </wsp:Policy>
    <types>...</types>
    <message>...</message>
    <portType>...</portType>

    <binding name="AddNumbersPortBinding" type="tns:AddNumbers">
       <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
       <operation name="addNumbers">
<wsp:PolicyReference URI="#AddNumbersPortBinding_addNumbers_WSAT_Policy" />
         <soap:operation soapAction="addNumbers" />
         <input>
<wsp:PolicyReference URI="#AddNumbersPortBinding_addNumbers_WSAT_Policy" />
             <soap:body use="literal" />
         </input>
         <output>
<wsp:PolicyReference URI="#AddNumbersPortBinding_addNumbers_WSAT_Policy" />
             <soap:body use="literal" />
         </output>
       </operation>
    </binding>
    <service>...</service>
<definitions>
```

16.2.2. Configuring from Java Class

To configure a web service transaction in a java class, add the following annotation in the web service endpoint implementation class.

```
@com.sun.xml.ws.api.tx.at.Transactional(version=com.sun.xml.ws.api.tx.at.Transactional.Version.WSAT10
)
```

The following is an example of defining a web service transaction in a java class.

Configuring a Web Service Transaction in a Java Class: <AddnumbersImpl.java>

```
@WebService
@com.sun.xml.ws.api.tx.at.Transactional(
       version=com.sun.xml.ws.api.tx.at.Transactional.Version.WSAT10)
public class AddNumbersImpl {
    ...
}
```

16.3. Client Configurations

There are no additional settings for web service transaction on the client side. The client applications that invoke the web service automatically provides a web service transaction environment at runtime by remotely reading the web service transaction policy from the WSDL.

16.4. Coordinator Service

To use a web service transaction, a coordinator service that tunes the transaction activities between participants must be deployed. A coordinator service is required for both server and client applications. Only one coordinator service must be deployed when a web service and a web service client are running on the same server.

The 'wstx-services.ear' file for coordinator service is in the following path.

```
JEUS_HOME/lib/systemapps
```

16.5. Web Service Transaction Example

Web service transaction implementation using a Java class is the same as the implementation of other basic JEUS 9 web services, except that it requires adding the @com.sun.xml.ws.api.tx.at.Transactional annotation.

Java Transaction API (JTA) is used to implement a web service transaction on a web service client.

Web Service Transaction: <AddNumbersClient.jsp>

```
InitialContext  ctx = new InitialContext();
UserTransaction utx = (UserTransaction) ctx.lookup("java:comp/UserTransaction");

AddNumbersImplService service = new AddNumbersImplService();
AddNumbersImpl port = service.getAddNumbersImplPort();

utx.begin();

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

```
utx.commit();
```

17. Web Service Security

This chapter describes security at transport-level and message-level, and how to apply them to JEUS web services.

17.1. Overview

The following are two ways to apply security to web services.

- Transport-level Security

  Maintains the security of a connection between client and web service by using SSL.

  Transport-level security via SSL secures the connection between a client and JEUS server. However, this secures only the connection itself. If there is an intermediary like a router or a message queue between the client and JEUS server, the intermediary may allow an unencrypted SOAP message to go through. Transport-level security cannot be applied to only a part of the message, but to the whole.

- Message-level Security

  Digitally signs or encrypts SOAP messages.

  Message-level security provides the strength of security through SSL as well as additional flexibility. Message-level security provides the strength of security through SSL as well as additional flexibility. Message-level security is an end-to-end security that maintains the security through multiple intermediaries when transmitting a message. In addition to securing the connection, it provides digital signature and encryption for a SOAP message. The digital signature and encryption can also be applied to only a part of the message.

17.2. Transport-level Security

Transport-level security for a web service uses SSL to secure a connection between a web service and a client application.

The following are the steps for transport-level security.

1. Configure SSL on JEUS server.

   There are no additional settings required for JEUS web service development. For more information about setting SSL on JEUS server, refer to "JEUS Web Engine Guide".

2. Configure SSL for a client application using these steps.

   a. Obtain the digital certificates. (Save the certificates to a local directory via the Internet Explorer, etc.)

   b. Store the certificates in the Keystore.

c. When creating a stub from WSDL using wsimport, or when executing the client to invoke a web service, configure the system property variables as in the following.

```
–Djavax.net.ssl.trustStore=keystore_name
```

```
–Djavax.net.ssl.trustStorePassword=keystore_password
```

d. To use the wsimport tool, additionally configure the following environment variable.

```
set WSIMPORT_OPTS=-Djavax.net.ssl.trustStore=keystore_name
      -Djavax.net.ssl.trustStorePassword=keystore_password
```

17.3. Message-level Security

The following are the web service security specifications for a JEUS web service.

- Web Service Security Policy
- Web Service Security
- Web Services Secure Conversation
- Web Service Trust

This section describes each specification definition and scenario.

17.3.1. Web Service Security Policy

The Web Service Security Policy (WS-Security Policy) specification defines assertions that describe how a message is secured and transmitted. The assertions are flexible enough to encompass policies for token, encryption, used mechanism, transport security, etc. The web service security policy is a key element for implementing web service message security.

JEUS web service provides a sample scenario of each message level security type for web services. The policy files, to which the web service security policy of each scenario was applied, can be helpful when applying various types of message security for web services.

17.3.2. Web Service Security

The Web Service Security (WS-Security) specification defines a standard SOAP extension group. It is used to implement a secure web service that preserves data integrity through XML signature and data confidentiality through XML encryption.

The following are web service security scenarios using the Web Service Security specification.

- Enhanced symmetric binding verification function through user name authentication
- Mutual Certificates Security
- SAML (Security Assertions Mark-up Language) authentication through SSL

17.3.2.1. Improving Symmetric Binding through Username Authentication

The following is an example of message security using symmetric binding. As already mentioned, in symmetric binding, both client and server use the same certificate for encrypted key and signature. This example uses the server's certificate. Since there is no method for the server to authenticate the client's identity, it also uses a username token.

The following describes a scenario using user name authentication and keystore settings.

- **Scenario**

  User name authentication scenario:

  1. The client encrypts and signs a request message by using the created symmetric key. To do this, it transmits the message with the symmetric key by using a public key of the server along with the username and password.
  2. The server decrypts the symmetric key by using the server's private key, and then uses the symmetric key to decode the client's request message and verify the signature. In addition, it authenticates the client by using the username and password.
  3. The server uses the same symmetric key to sign and encrypt a response message, and sends the message without the symmetric key.
  4. The client uses the symmetric key in its possession to decode the encrypted response message and verifies the signature with the server's public key.

- **Keystore setting**

  The following are the keystore settings.

Type	Description
Client	Keystore that contains the public key of the server.
Server	Keystore that contains the private key of the server.

17.3.2.2. Mutual Certificates Security

The following example shows message security using asymmetric binding and mutual certificates. Unlike symmetric binding, in asymmetric binding, the client and server use different certificates for encrypted key and signature. They sign with their own private keys and transmit certificate information, so that they can authenticate one another when verifying the signatures.

The following describes a scenario of mutual certificates security and keystore settings.

- **Scenario**

Mutual certificates security scenario:

1. The client signs a request message with its private key, and creates a symmetric key to encrypt the message. The client, then, encrypts the symmetric key by using the server’s public key, and transmits a request message with the symmetric key to the server along with the client certificate for authentication.
2. The server decodes the symmetric key by using its private key, and decodes the encrypted message. It authenticates the certificate and verifies the signature with the client’s public key.
3. The server uses its own private key to sign the response message and creates a symmetric key to encrypt the message. Then, it uses the client’s public key to encrypt the symmetric key and sends it in a response message to the client.
4. The client decodes the symmetric key by using its private key, and decodes the encrypted response message. It also verifies the signature by using the server’s public key.

- **Keystore setting**

  The following are the keystore settings.

Type	Description
Client	Keystore that contains the public key of the server.
Server	Keystore that contains the private key of the server.

17.3.2.3. SAML Authentication through SSL

This section describes how to include the Security Assertions Mark-up Language (SAML) token in the SOAP message header by creating and transmitting the SAML token as text through a handler.

To use the SAML token for the actual web service, the SAML token framework, which creates the SAML token, is required.

The development and increase in usage of web services increased the need for exchanging security and authentication information between various web applications resulting in a need for exchange standards. SAML(Security Assertions Mark-up Language) is a language that was developed for this purpose.

SAML provides the following functionalities.

- Provides XML format for user security information and a format for requesting and transmitting such information.
- Defines how to use such messages in a protocol such as SOAP.
- Specifies how to exchange the message in general cases such as web SSO.
- Supports many mechanisms for protecting personal information. For instance, it determines user properties without exporting the user’s identity.
- Provides a way to process ID information in a format provided by the commonly used technologies such as Unix, Microsoft Windows, X.509, LDAP, DCE and XCML.

- Provides a function for communicating with the SAML option supported by the system that formulates the metadata schema.

The following is a scenario of SAML authorization over SSL.

- **Scenario**

  1. The client includes the SAML token in the SOAP message header, and transmits a message through the SSL setting.
  2. The server interprets the SAML token included in the SOAP message header, and transmits a response message.

Message exchanges between the client and server is all done through the SSL setting of the JEUS server.

17.3.3. Web Services Secure Conversation

The Web Services Secure Conversation (WS-Secure Conversation) specification defines the creation and sharing of secure context between a service provider and client. The secure context is used to reduce the overheads during message exchanges. The specification defines the standards for better message level security and more efficient exchanges of multiple messages.

JEUS web services follow the WS-SecureConversation and provides the basic mechanism that defines the security method for multi-message exchanges, and helps to establish the context with a more efficient key or a new key element. This can help improve the overall performance and security for message exchanges.

In symmetric binding, an encrypted key needs to be created whenever a message is exchanged, which may not be suitable for multi-message exchanges. In this example, SecureContext is established for both sides by asymmetric binding during the handshaking process before exchanging the first message through the Secure Conversation. When exchanging messages multiple times, the established SecureContext is used for signature and encryption like in symmetric binding. The authentication process is performed during the first SecureContext establishment process through asymmetric binding.

The following describes a scenario of web services secure conversation and keystore settings.

- **Scenario**

  Web services secure conversation scenario:

  1. The client signs its own certificate and SecureContext information by using the certificate, and encrypts them with the server's public key to transmit to the server. (Asymmetric binding)
  2. After decoding the client's message with its private key and verifying the client's signature, the server responds with a message after establishing SecureContext. (Asymmetric binding)
  3. When exchanging messages, the client and server use the SecureContext information to encrypt and sign the message. (Symmetric binding)

- **Keystore setting**

  The following are the keystore settings.

Type	Description
Client	Keystore that contains the client's private key and the server's public key.
Server	Keystore that contains the server's private key and the client's public key.

17.3.4. Web Service Trust

This section describes an example of using WS-Trust for authentication. The certificate between different tokens, which is an untrusted relationship, is converted into the SAML token through STS, and the service provider who cannot trust the service consumer can authenticate it through STS.

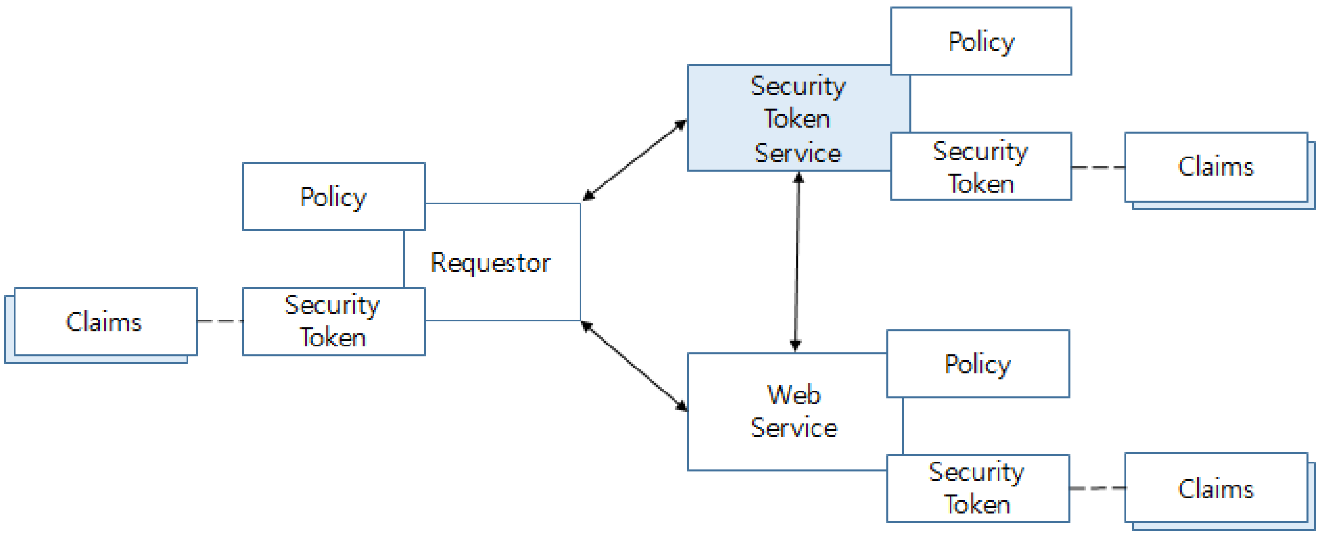
To exchange messages using web service security, the service and client should have already agreed upon a security token that is used for sharing security information. Otherwise, a trust relationship must be established between the parties before exchanging messages. This is referred as Web Service Trust, which is supported by JEUS web services.

Web Service Trust can be used when a server and a client cannot authenticate each other when they each use a different security token or each security token is from a different domain. If they cannot authenticate each other due to different security tokens, another web service, called **STS(Security Token Service)**, can mediate in deciding their authentication roles.

In a web service trust relationship:

- A client and the server have no direct trust relationship with each other.
- A client and an STS have a trust relationship with each other.
- An STS and the server have a trust relationship with each other.

As shown in the following figure, STS dynamically issues the security token to the requester, so that the web service can authenticate the requester before communicating. The requester transmits a message with the issued security token to the web service, so that it can authenticate(trust) the requester.



WS-Trust

(*Source: http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html)

The following is a scenario of web service trust and keystore settings.

- **Scenario**

  Web service trust scenario:

  1. The client requests STS to issue an SAML certificate for the server, to where the client will transmit the actual message with the client's certificate, and a username and password.
  2. STS issues the SAML certificate after authenticating the client.
  3. The client transmits a message with the SAML certificate to the server.
  4. The server trusts the client through the certificate, and processes the requested business logic.

- **Keystore setting**

  The following are the keystore settings.

Type	Description
Client	• Keystore that contains the client's private key and the server's public key • Keystore that contains the client's private key and the STS server's public key
Server	Keystore that contains the server's private key and the client's public key
STS server	Keystore that contains the server's private key and the client's public key

17.4. Configuring Message-level Security

This section introduces common settings applied to the example scenarios of message level security for JEUS web services, and describes how to configure them in each scenario. Refer to the examples to see how they actually work.

The same business logic for the client and web service is used in the message level security configuration example for each scenario.

17.4.1. Common Configurations

Common settings applied to example scenarios of message level security in JEUS web service are divided into server and client settings.

17.4.1.1. Server Configurations

The following are some common security settings applied to the 'endpoint-policy-subject', 'operation-policy-subject' and 'input(output)-message-policy-subject' elements.

- **Configuring 'keystore' and 'truststore' elements**

  The 'keystore' and 'truststore' elements below the 'security-policy' element must be actively set according to the 'security-binding' settings below the 'security-policy' element.

  To set a keystore in the 'keystore-filename' element, set the path of the actual keystore file with an absolute or relative path. When using a relative path, the keystore file should be placed in the 'META-INF' directory under the 'classes' directory, which contains the service implementation classes.

  Configuring Server Message-Level Security (1): <jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" version="7.0">
    ...
    <xs:complexType name="keyTruststoreType">
        <xs:choice>
            <xs:element name="keystore-file" type="keyTruststoreFileType" />
            <xs:element name="keystore-callbackhandler" type="xs:string" />
        </xs:choice>
    </xs:complexType>
    <xs:complexType name="keyTruststoreFileType">
        <xs:sequence>
            <xs:element name="alias" type="xs:string" />
            <xs:element name="key-type" type="xs:string" minOccurs="0"
            default="JKS" />
            <xs:element name="keystore-password" type="xs:string" />
            <xs:element name="keystore-filename" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

- **Configuring the 'include-token' attribute**

The 'include-token' attribute is used to configure whether to include a token in the message.

The following is an example of the X509 token.

Configuring Server Message-Level Security (2): <jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" version="7.0">
    ...
    <xs:complexType name="x509TokenType">
        <xs:sequence>
            <xs:element name="include-token" type="xs:boolean" minOccurs="0"
            default="false" />
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

For more information about other element settings, refer to the 'jeus-webservices-config.xsd' schema, "JEUS Reference Guide" or sample examples.

Implementing a Web Service from Java Class

To implement web service message security using a Java class, create a web service by using the '-policy' option of the **wsgen** tool, as shown in the following example.

```
$ wsgen fromjava.server.AddNumbersImpl -d web/WEB-INF -policy service-config.xml
```

To configure the service-config.xml file used as an argument of the '-policy' option, foreknowledge about the contents of 'jeus-webservices-config.xsd', which is the JEUS security policy schema for 'service-config.xml', is required.

JEUS security policy schema can set security settings in three elements, 'endpoint-policy-subject', 'operation-policy-subject,' and 'input(output)-message-policy-subject'.

Configuring Server Message Level-Security with a Java Class (1): <jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" version="7.0">
    ...
    <xs:element name="web-services-config" type="web-services-configType" />
    <xs:complexType name="web-services-configType">
        <xs:choice>
            ...
            <xs:element name="policy" type="policy-configType"
            maxOccurs="unbounded" />
        </xs:choice>
    </xs:complexType>
    <xs:complexType name="policy-configType">
        <xs:sequence>
            <xs:element name="endpoint-policy-subject"
            type="endpointPolicySubjectType" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="endpointPolicySubjectType">
        <xs:sequence>
            ...
            <xs:element name="security-policy"
            type="endpointSecurityPolicyType" minOccurs="0" />
            <xs:element name="operation-policy-subject"
            type="operationPolicySubjectType" minOccurs="0"
            maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="operationPolicySubjectType">
        <xs:sequence>
            <xs:element name="security-policy"
            type="operationSecurityPolicyType" minOccurs="0" />
            ...
            <xs:element name="input-message-policy-subject"
            type="messagePolicySubjectType" minOccurs="0" />
            <xs:element name="output-message-policy-subject"
            type="messagePolicySubjectType" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="messagePolicySubjectType">
        <xs:sequence>
            <xs:element name="security-policy" type="messageSecurityPolicyType"
            minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

The following shows how to configure these three elements.

- **Configuring the 'endpoint-policy-subject' element**

  Security settings can be configured for each service implementation class. Various settings related to message security, such as security binding (asymmetry and symmetry), signature or encryption, security token, and WS-Security versions, can be configured.

  Configuring Server Message-Level Security with a Java Class (2): <jeus-webservices-config.xsd>

  ```
  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
  targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
  attributeFormDefault="unqualified" version="7.0">
      ...
      <xs:complexType name="endpointSecurityPolicyType">
          <xs:sequence>
  ```

```
            <xs:element name="security-binding" type="securityBindingType"
            minOccurs="0" />
            <xs:element name="token" type="supportingTokenType" minOccurs="0" />
            <xs:element name="protection" type="protectionType" minOccurs="0" />
            ...
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

There are three kinds of security settings for the 'endpoint-policy-subject' element.

- security-binding

  As shown in the following, the security settings for the security-binding element can be set to either 'transport-binding,' 'symmetric-binding,' or 'asymmetric-binding'.

  Configuring Server Message-Level Security with a Java Class (3): <jeus-webservices-config.xsd>

  ```
  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
  targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
  attributeFormDefault="unqualified" version="7.0">
      ...
      <xs:complexType name="securityBindingType">
          <xs:sequence>
              <xs:choice>
                  <xs:element name="transport-binding"
                  type="transportBindingType" />
                  <xs:element name="symmetric-binding"
                  type="symmetricBindingType" />
                  <xs:element name="asymmetric-binding"
                  type="asymmetricBindingType" />
              </xs:choice>
          </xs:sequence>
      </xs:complexType>
      ...
  </xs:schema>
  ```

  - 'transport-binding' element

    The security setting for the 'transport-binding' element denotes message protection using a transport protocol other than the WS-Security specification (e.g., HTTPS).

    Configuring Server Message-Level Security with a Java Class (4): <jeus-webservices-config.xsd>

    ```
    <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
    <xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
    targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    attributeFormDefault="unqualified" version="7.0">    ...
        <xs:complexType name="transportBindingType">
            <xs:sequence>
                <xs:element name="transport-token" type="tokenType" />
    ```

```
            <xs:element name="algorithm-suite" type="algorithmSuiteType"
            minOccurs="0" default="Basic128" />
            <xs:element name="layout" type="layoutType" minOccurs="0"
            default="Lax" />
            <xs:element name="timestamp" type="xs:boolean" minOccurs="0"
            default="true" />
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

- 'symmetric-binding' element

  The security settings for the 'symmetric-binding' element is used to protect a message through symmetric binding according to the WS-security specification.

  Symmetric binding is when the same token is used to encrypt or decrypt a symmetric key. It is used to encrypt or decrypt a message and is transmitted as encrypted below an additional element, EncryptedKey, of the SOAP message header. This means that the same token is used to sign and verify the signature. The child element, 'protection-token', indicates that the same token is used to encrypt and sign a message.

  Configuring Server Message-Level Security with a Java Class (5): <jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" version="7.0">
    ...
    <xs:complexType name="symmetricBindingType">
        <xs:sequence>
            <xs:element name="protection-token" type="tokenType" />
            <xs:element name="algorithm-suite" type="algorithmSuiteType"
            minOccurs="0" default="Basic128" />
            <xs:element name="layout" type="layoutType" minOccurs="0"
            default="Lax" />
            <xs:element name="timestamp" type="xs:boolean" minOccurs="0"
            default="true" />
            <xs:element name="encrypt-signature" type="xs:boolean"
            minOccurs="0" default="false" />
            <xs:element name="encrypt-before-siging" type="xs:boolean"
            minOccurs="0" default="false" />
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

- 'asymmetric-binding' element

  The security settings for the 'symmetric-binding' element is used to protect a message through symmetric binding according to the WS-security specification.

  A symmetric binding is when a different token is used to encrypt and decrypt a symmetric key. A symmetric key is used to encrypt or decrypt a message and is transmitted as encrypted below an additional element, EncryptedKey, of the SOAP message header. This means that a different token is used to sign and verify the signature.

Configuring Server Message-Level Security with a Java Class (6): <jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" version="7.0">
    ...
    <xs:complexType name="asymmetricBindingType">
        <xs:sequence>
            <xs:element name="initiator-token" type="tokenType" />
            <xs:element name="recipient-token" type="tokenType" />
            <xs:element name="algorithm-suite" type="algorithmSuiteType"
            minOccurs="0" default="Basic128" />
            <xs:element name="layout" type="layoutType" minOccurs="0"
            default="Strict" />
            <xs:element name="timestamp" type="xs:boolean" minOccurs="0"
            default="true" />
            <xs:element name="encrypt-before-siging" type="xs:boolean"
            minOccurs="0" default="false" />
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

The following is a description of child elements.

Element	Description
initiator-token	A token is used by the client to sign a message and also to decrypt an encrypted message from the server. The token is used by the server to encrypt a message and also to verify the client signature.
recipient-token	A token is used by the server to sign a message and also to decrypt an encrypted message from the client. The token is used by the client to encrypt a message and also to verify the server signature.

◦ token

One of the child elements can be selected for the security setting of the token element.

Configuring Server Message-Level Security with a Java Class (7): <jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
```

```
attributeFormDefault="unqualified" version="7.0">
    ...
<xs:complexType name="supportingTokenType">
        <xs:choice>
            <xs:element name="supporting-token" type="tokenType" />
            <xs:element name="signed-supporting-token" type="tokenType" />
            <xs:element name="endorsing-supporting-token" type="tokenType" />
            <xs:element name="signed-endorsing-supporting-token"
            type="tokenType" />
        </xs:choice>
    </xs:complexType>
    ...
</xs:schema>
```

The following is a description of child elements.

Element	Description
supporting-token	This token value is included in the message header, and can be used to encrypt or decrypt other parts of the message.
signed-supporting-token	Same as 'supporting-token'. In addition, this token can be signed.
endorsing-supporting-token	Used to configure a token which is used to resign a signature. Signs the entire 'Signature' element.
signed-endorsing-supporting-token	Same as 'endorsing-supporting-token'. In addition, this token is signed through the original signature (mutual signing).

- protection

  One of the child elements can be selected for the security setting of the protection element.

  Configuring Server Message Level Security with a Java Class (8): <jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" version="7.0">
    ...
<xs:complexType name="protectionType">
        <xs:sequence>
            <xs:element name="signed-part" type="protectionPartType"
            minOccurs="0" />
            <xs:element name="encrypted-part" type="protectionPartType"
            minOccurs="0" />
            <xs:element name="signed-element" type="xs:string"
            minOccurs="0" />
            <xs:element name="encrypted-element" type="xs:string"
            minOccurs="0">
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

The following is a description of child elements.

Element	Description
signed-part	Part to sign or verify a signature for.
encrypted-part	Part to encrypt or decrypt.
signed-element	Part to sign or verify a signature for. This is useful for signing a part of an element. To use this element, the 'disable-streaming-security' element must be set to 'true'. For more information, refer to the example.
encrypted-element	Part of an element to encrypt or decrypt. This is useful for encrypting a part of an element. To use this element, the 'disable-streaming-security' element must be set to 'true'. For more information, refer to the example.

- **Configuring the 'operation-policy-subject' element**

  Security settings can be configured for each service implementation class method. Additionally, signature, encryption, and security tokens can be configured.

  The 'operation-policy-subject' element, which is used to configure security settings for a service implementation class method, is largely divided into the following two types.

  Configuring Server Message-Level Security with a Java Class (9): <jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" version="7.0">
    ...
    <xs:complexType name="operationSecurityPolicyType">
        <xs:sequence>
            <xs:element name="supporting-token" type="supportingTokenType"
            minOccurs="0" />
            <xs:element name="protection" type="protectionType" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

  The following is a description of child elements.

Element	Description
supporting-token	Same as the 'token' element inside the 'endpoint-policy-subject' element, but this setting only applies to a method.
protection	Same as the 'protection' element inside the 'endpoint-policy-subject' element, but this setting only applies to a method.

- **Configuring the 'input(output)-message-policy-subject' element**

  Security setting is configured for each parameter and return value of service implementation class methods. Additionally, signature, encryption, and security tokens can be configured.

  The following is the security setting for each parameter and return value of service implementation class methods. There are two kinds of security settings for the 'input(output)-message-policy-subject' element.

  Configuring Server Message-Level Security with a Java Class (10): < jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns="http://www.tmaxsoft.com/xml/ns/jeus"
targetNamespace="http://www.tmaxsoft.com/xml/ns/jeus"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" version="7.0">
    ...
    <xs:complexType name="messageSecurityPolicyType">
        <xs:sequence>
            <xs:element name="supporting-token" type="supportingTokenType"
            minOccurs="0" />
            <xs:element name="protection" type="protectionType" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

  The following is a description of child elements.

Element	Description
supporting-token	Same as the 'token' element inside the 'endpoint-policy-subject' element, but this setting only applies to parameters and return values of the method.
protection	Same as the 'protection' element inside the 'endpoint-policy-subject' element, but this setting only applies to parameters and return values of the method.

Implementing a Web Service from WSDL

To implement a web service from WSDL, configure the message security policy directly in the WSDL document, and create a web service by using the **wsimport** tool. A simple web service message security can be implemented through WSDL by understanding the WS-SecurityPolicy specification, which is below the WS-policy specification, and applying it to WSDL.

For more information about the WS-SecurityPolicy, refer to WS-SecurityPolicy 1.2 Specification.

The following is an example of WSDL, with message security configured, that implements a web service. To set a keystore in the 'keystore-filename' element, set the path of the actual keystore file with an absolute or relative path. When using a relative path, the keystore file should be placed in the 'META-INF' directory under the 'classes' directory, which contains the service implementation classes.

Configuring WSDL Message-Level Security: <jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions
...
targetNamespace="http://tmax.com/" name="NewWebServiceService">
<ns1:Policy xmlns:ns1="http://schemas.xmlsoap.org/ws/2004/09/policy"
 wsu:Id="NewWebServicePortBindingPolicy">
  <ns1:ExactlyOne>
    <ns1:All>
      <ns7:SymmetricBindingxmlns:ns7="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <ns1:Policy>
            <ns1:ExactlyOne>
              <ns1:All>
                <ns7:AlgorithmSuite>
                  <ns1:Policy>
                    <ns1:ExactlyOne>
                      <ns1:All>
                        <ns7:TripleDes />
                      </ns1:All>
                    </ns1:ExactlyOne>
                  </ns1:Policy>
                </ns7:AlgorithmSuite>
                <ns7:IncludeTimestamp />
                <ns7:Layout>
                  <ns1:Policy>
                    <ns1:ExactlyOne>
                      <ns1:All>
                        <ns7:Strict />
                      </ns1:All>
                    </ns1:ExactlyOne>
                  </ns1:Policy>
                </ns7:Layout>
                <ns7:OnlySignEntireHeadersAndBody />
                <ns7:ProtectionToken>
                  <ns1:Policy>
                    <ns1:ExactlyOne>
                      <ns1:All>
                        <ns7:X509Tokenns7:IncludeToken=
            "http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never">
                          <ns1:Policy>
                            <ns1:ExactlyOne>
                              <ns1:All>
                              <ns7:WssX509V3Token10/>
                              </ns1:All>
                            </ns1:ExactlyOne>
                          </ns1:Policy>
                        </ns7:X509Token>
                      </ns1:All>
                    </ns1:ExactlyOne>
                  </ns1:Policy>
                </ns7:ProtectionToken>
              </ns1:All>
            </ns1:ExactlyOne>
          </ns1:Policy>
      </ns7:SymmetricBinding>
      <ns8:Wss11xmlns:ns8="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <ns1:Policy>
              <ns1:ExactlyOne>
                  <ns1:All>
                      <ns8:MustSupportRefEncryptedKey />
                      <ns8:MustSupportRefIssuerSerial />
                      <ns8:MustSupportRefThumbprint />
                  </ns1:All>
              </ns1:ExactlyOne>
          </ns1:Policy>
      </ns8:Wss11>
      <sc:KeyStore wspp:visibility="private"
      alias="xws-security-server"
      storepass="changeit" type="JKS"
      location="keystore.jks" />
      <sc:TrustStore wspp:visibility="private"
      peeralias="xws-security-client"
      storepass="changeit" type="JKS"
      location="cacerts.jks" />
      <sc:ValidatorConfigurationxmlns:sc=
            "http://schemas.sun.com/2006/03/wss/server">
        <sc:Validator name="usernameValidator"
             classname="com.tmax.UsernamePasswordValidator" />
      </sc:ValidatorConfiguration>
      <ns9:UsingAddressingxmlns:ns9="http://www.w3.org/2006/05/addressing/wsdl" />
    </ns1:All>
  </ns1:ExactlyOne>
</ns1:Policy>
<ns10:Policy xmlns:ns10="http://schemas.xmlsoap.org/ws/2004/09/policy"
       wsu:Id="NewWebServicePortBinding_add_Input_Policy">
  <ns10:ExactlyOne>
    <ns10:All>
      <ns11:EncryptedPartsxmlns:ns11=
           "http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <ns11:Body />
      </ns11:EncryptedParts>
      <ns12:SignedPartsxmlns:ns12=
           "http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <ns12:Body />
        <ns12:Header Name="ReplyTo"
            Namespace="http://www.w3.org/2005/08/addressing" />
        <ns12:HeaderNamespace="http://www.w3.org/2005/08/addressing" Name="To"/>
        <ns12:Header Name="From"
            Namespace="http://www.w3.org/2005/08/addressing"/>
        <ns12:Header Name="MessageId"
            Namespace="http://www.w3.org/2005/08/addressing"/>
        <ns12:Header Name="FaultTo"
            Namespace="http://www.w3.org/2005/08/addressing"/>
        <ns12:Header Name="Action"
            Namespace="http://www.w3.org/2005/08/addressing"/>
        <ns12:Header Name="RelatesTo"
            Namespace="http://www.w3.org/2005/08/addressing"/>
      </ns12:SignedParts>
      <ns13:SignedSupportingTokensxmlns:ns13=
         "http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <ns10:Policy>
          <ns10:ExactlyOne>
            <ns10:All>
             <ns13:UsernameToken ns13:
               IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/
               IncludeToken/AlwaysToRecipient">
               <ns10:Policy>
                 <ns10:ExactlyOne>
                   <ns10:All>
                   <ns13:WssUsernameToken10/>
                   </ns10:All>
                 </ns10:ExactlyOne>
               </ns10:Policy>
             </ns13:UsernameToken>
            </ns10:All>
          </ns10:ExactlyOne>
        </ns10:Policy>
      </ns13:SignedSupportingTokens>
    </ns10:All>
  </ns10:ExactlyOne>
</ns10:Policy>
<ns14:Policy xmlns:ns14="http://schemas.xmlsoap.org/ws/2004/09/policy"
  wsu:Id="NewWebServicePortBinding_add_Output_Policy">
  <ns14:ExactlyOne>
    <ns14:All>
      <ns15:EncryptedPartsxmlns:ns15="http://schemas.xmlsoap.org/ws/
        2005/07/securitypolicy">
        <ns15:Body />
      </ns15:EncryptedParts>
      <ns16:SignedPartsxmlns:ns16="http://schemas.xmlsoap.org/ws/
        2005/07/securitypolicy">
        <ns16:Body />
        <ns16:Header Name="ReplyTo"
              Namespace="http://www.w3.org/2005/08/addressing"/>
        <ns16:HeaderNamespace="http://www.w3.org/2005/08/addressing"
              Name="To"/>
        <ns16:Header Name="From"
              Namespace="http://www.w3.org/2005/08/addressing"/>
        <ns16:Header Name="MessageId"
              Namespace="http://www.w3.org/2005/08/addressing"/>
        <ns16:Header Name="FaultTo"
              Namespace="http://www.w3.org/2005/08/addressing"/>
        <ns16:Header Name="Action"
              Namespace="http://www.w3.org/2005/08/addressing"/>
        <ns16:Header Name="RelatesTo"
              Namespace="http://www.w3.org/2005/08/addressing"/>
      </ns16:SignedParts>
    </ns14:All>
  </ns14:ExactlyOne>
</ns14:Policy>
<types>
    ...
</types>
<message name="add">
    ...
</message>
<portType name="NewWebService">
    ...
</portType>
<binding name="NewWebServicePortBinding" type="tns:NewWebService">
  <ns17:PolicyReferencexmlns:ns17="http://schemas.xmlsoap.org/ws/2004/09/policy"
       URI="#NewWebServicePortBindingPolicy" />
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
```

```
  <operation name="add">
    <soap:operation soapAction="" />
    <input>
      <ns18:PolicyReferencexmlns:ns18="http://schemas.xmlsoap.org/ws/2004/09/policy"
          URI="#NewWebServicePortBinding_add_Input_Policy" />
      <soap:body use="literal" />
    </input>
    <output>
      <ns19:PolicyReferencexmlns:ns19="http://schemas.xmlsoap.org/ws/2004/09/policy"
         URI="#NewWebServicePortBinding_add_Output_Policy" />
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
<service name="NewWebServiceService">
 <port name="NewWebServicePort" binding="tns:NewWebServicePortBinding">
   <soap:addresslocation="http://localhost:8088/usernameAuthentication_war/
   NewWebServiceService"/>
 </port>
 </service>
</definitions>
```

17.4.1.2. Client Configurations

For a web service client in which message security is configured, a keystore and a callback handler must be additionally configured in the wsit-client.xml file. Other message security settings are automatically configured by reading the message security policy from the server-side WSDL at runtime.

To set a keystore in the 'keystore-filename' element, set the path of the actual keystore file with an absolute or relative path. When using a relative path, the keystore file should be placed in the 'META-INF' directory under the 'classes' directory, which contains the service implementation classes.

Configuring Client Message-Level Security: <jeus-webservices-config.xsd>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
...
targetNamespace="http://server.fromjava/">
  <wsp:UsingPolicy />
    <wsp:Policy wsu:Id="TmaxBP0">
        <wsp:ExactlyOne>
            <wsp:All>
                <CallbackHandlerConfiguration
                xmlns="http://schemas.sun.com/2006/03/wss/client">
                    <CallbackHandler default="user_jeus" name="usernameHandler" />
                    <CallbackHandler default="password_jeus"
                    name="passwordHandler" />
                </CallbackHandlerConfiguration>
                <TrustStore location="cacerts.jks" peeralias="xws-security-server"
                storepass="changeit" type="JKS"
                xmlns:ns0="http://java.sun.com/xml/ns/wsit/policy"
                ns0:visibility="private"
                xmlns="http://schemas.sun.com/2006/03/wss/client" />
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>
    <types>
        ...
    </types>
    <message name="addNumbers">
        ...
    </message>
    <portType name="AddNumbersImpl">
        ...
    </portType>
    <binding name="AddNumbersImplPortBinding" type="tns:AddNumbersImpl">
        <wsp:PolicyReference URI="#TmaxBP0" />
        <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
        <operation name="addNumbers">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>
    <service name="AddNumbersImplService">
        <port binding="tns:AddNumbersImplPortBinding" name="AddNumbersImplPort">
            <soap:address
location="http://localhost:8088/DocLitEchoService/AddNumbersImplService" />
        </port>
    </service>
</definitions>
```

17.4.2. Improving Symmetric Binding through Username Authentication

This section describes how to improve symmetric binding through user name authentication for server and client applications.

17.4.2.1. Server Configurations

There are two kinds of web service settings on the server side, the web service configuration file, 'service-config.xml', and username validator setting.

Creating Web Service Configuration File

The following is an example of creating a web service configuration file.

Web Service Configuration File: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
```

```
    <policy>
        <endpoint-policy-subject>
            <security-policy>
                <security-binding>
                    <symmetric-binding>
                        <protection-token>
                            <x509-token>
                                <include-token>true</include-token>
                            </x509-token>
                        </protection-token>
                    </symmetric-binding>
                </security-binding>
                <token>
                    <signed-supporting-token>
                        <username-token>
                            <username-password-validator>
                                fromjava.server.UsernamePasswordValidator
                            </username-password-validator>
                            <include-token>true</include-token>
                        </username-token>
                    </signed-supporting-token>
                </token>
                <protection>
                    <signed-part>...</signed-part>
                    <encrypted-part>...</encrypted-part>
                </protection>
                <wss-version>11</wss-version>
                <keystore>...</keystore>
            </security-policy>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

Creating a Username Validator Class

Validate the username and password of the user who used a username token to access the service by implementing the class, com.sun.xml.wss.impl.callback.PasswordValidationCallback.PasswordValidator.

Username Validator Class: <UsernamePasswordValidator.java>

```
public class UsernamePasswordValidator implements
        PasswordValidationCallback.PasswordValidator {

   public boolean validate(PasswordValidationCallback.Request request)
           throws PasswordValidationCallback.PasswordValidationException {
       ...
       return false;
   }

   private boolean validateUserFromDB(String username, String password) {
       ...
       return false;
   }
}
```

17.4.2.2. Client Configurations

To configure a client-side web service, a keystore (truststore) must also be configured in the wsit-client.xml file.

Configuring a Keystore (Truststore) and a Username Handler

Configure the following settings in the wsit-client.xml file.

Configuring Keystore (Truststore) and Username Handler: <wsit-client.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
...
targetNamespace="http://server.fromjava/">
    <wsp:UsingPolicy />
    <wsp:Policy wsu:Id="TmaxBP0">
        <wsp:ExactlyOne>
            <wsp:All>
                <CallbackHandlerConfiguration
                xmlns="http://schemas.sun.com/2006/03/wss/client">
                    <CallbackHandler default="user_jeus" name="usernameHandler" />
                    <CallbackHandler default="password_jeus"
                    name="passwordHandler" />
                </CallbackHandlerConfiguration>
                <TrustStore location="cacerts.jks" peeralias="xws-security-server"
                storepass="changeit" type="JKS"
                xmlns:ns0="http://java.sun.com/xml/ns/wsit/policy"
                ns0:visibility="private"
                xmlns="http://schemas.sun.com/2006/03/wss/client" />
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>
    ...
    <binding name="AddNumbersImplPortBinding" type="tns:AddNumbersImpl">
        <wsp:PolicyReference URI="#TmaxBP0" />
        ...
    </binding>
    ...
</definitions>
```

Creating a Username Handler Class

Various settings can be configured by implementing the javax.security.auth.callback.CallbackHandler interface and transmitting each Callback to the CallbackHandler. The settings are configured according to the policy of the transmitted callback type.

The CallbackHandler interface sets a username and password for the callback by using the transmitted callbacks, NameCallback and PasswordCallback, from the previous example.

In the following example, a username and password are already specified in the class file, but the user can also enter the values using various reader objects as needed.

Username Handler Class: <UsernamePasswordCallbackHandler.java>

```
public class UsernamePasswordCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            Callback callback = callbacks[i];
            if (callback instanceof NameCallback) {
                ((NameCallback) callback).setName("user_jeus");
            } else if (callback instanceof PasswordCallback) {
                ((PasswordCallback) callback).setPassword((
                        new String("password_jeus")).toCharArray());
            } else {
                throw new UnsupportedCallbackException(callback,
                        "Unknow callback for username or password");
            }
        }
    }
}
```

17.4.3. Mutual Certificates Security

This section explains how to configure mutual certificates security in the server and client applications.

17.4.3.1. Server Configurations

A web service on the server side can be configured in the 'service-config.xml', the web service configuration file.

Configuring Server-Side Mutual Certificates Security: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
        <endpoint-policy-subject>
            <security-policy>
                <security-binding>
                    <asymmetric-binding>
                        <initiator-token>
                            <x509-token>
                                <include-token>true</include-token>
                            </x509-token>
                        </initiator-token>
                        <recipient-token>
                            <x509-token>
                                <include-token>false</include-token>
                            </x509-token>
                        </recipient-token>
                    </asymmetric-binding>
                </security-binding>
                <protection>
                    <signed-part>...</signed-part>
                    <encrypted-part>...</encrypted-part>
                </protection>
                <keystore>...</keystore>
                <truststore>...</truststore>
            </security-policy>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

17.4.3.2. Client Configurations

To configure a client-side web service, a keystore (truststore) must also be configured in the wsit-client.xml file.

Configuring Client-Side Mutual Certificates Security: <wsit-client.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
...
targetNamespace="http://server.fromjava/">
    <wsp:UsingPolicy/>
    <wsp:Policy wsu:Id="TmaxBP0">
        <wsp:ExactlyOne>
            <wsp:All>
                <KeyStore
                    alias="xws-security-client"
                    location="keystore.jks"
                    storepass="changeit" type="JKS"
                    xmlns:ns0="http://java.sun.com/xml/ns/wsit/policy"
                    ns0:visibility="private"
                    xmlns="http://schemas.sun.com/2006/03/wss/client"/>
                <TrustStore
                    location="cacerts.jks"
                    peeralias="xws-security-server"
                    storepass="changeit"
                    type="JKS"
                    xmlns:ns0="http://java.sun.com/xml/ns/wsit/policy"
                    ns0:visibility="private"
                    xmlns="http://schemas.sun.com/2006/03/wss/client"/>
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>
    ...
    <binding name="AddNumbersImplPortBinding" type="tns:AddNumbersImpl">
        <wsp:PolicyReference URI="#TmaxBP0"/>
        ...
    </binding>
    ...
</definitions>
```

17.4.4. SAML Authentication through SSL

This section describes how to configure SAML authentication over SSL on the server and client.

17.4.4.1. Server Configurations

A web service on the server side can be configured in the 'service-config.xml', the web service configuration file.

SAML Authentication through SSL: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
        <endpoint-policy-subject>
            <security-policy>
                <security-binding>
                    <transport-binding>
                        <transport-token>
                            <https-token/>
                        </transport-token>
                    </transport-binding>
                </security-binding>
            </security-policy>
            <operation-policy-subject>
                <operation-java-name>addNumbers</operation-java-name>
                <input-message-policy-subject>
                    <security-policy>
                        <supporting-token>
                            <signed-supporting-token>
                                <saml-token>
                                </saml-token>
                            </signed-supporting-token>
                        </supporting-token>
                    </security-policy>
                </input-message-policy-subject>
            </operation-policy-subject>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

17.4.4.2. Client Configurations

For the client-side web service settings, a handler class of the SAML message must be additionally configured in the wsit-client.xml file.

Configuring a Handler Class to Process SAML Messages

Configure a callback handler class to process SAML messages in wsit-client.xml as in the following example.

Configuring a Callback Handler Class: <wsit-client.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
...
targetNamespace="http://server.fromjava/">
    <ns5:Policy xmlns:ns5="http://schemas.xmlsoap.org/ws/2004/09/policy"
    wsu:Id="TmaxBO1TmaxIn1">
        <ns5:ExactlyOne>
            <ns5:All>
                <CallbackHandlerConfiguration
                xmlns="http://schemas.sun.com/2006/03/wss/client">
                    <CallbackHandler classname="xwss.saml.SamlCallbackHandler"
                    name="samlHandler" />
                </CallbackHandlerConfiguration>
            </ns5:All>
        </ns5:ExactlyOne>
    </ns5:Policy>
    ...
    <binding name="AddNumbersImplPortBinding" type="tns:AddNumbersImpl">
        ...
        <operation name="addNumbers">
            <soap:operation soapAction="" />
            <input>
                <ns10:PolicyReference
                xmlns:ns10="http://schemas.xmlsoap.org/ws/2004/09/policy"
                URI="#TmaxBO1TmaxIn1" />
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>
    ...
</definitions>
```

Creating a Handler Class to Process SAML Messages

In order to include information in the SOAP request message according to the SAML standard, the example uses SamlCallbackHandler class which implements javax.security.auth.callback.CallbackHandler. SamlCallbackHandler stores the required assertions according to the callback type.

The following is the SamlCallbackHandler class used in the example.

Creating a Callback Handler Class: <SamlCallbackHandler.java>

```
public class SamlCallbackHandler implements CallbackHandler {
  ...
  public void handle(Callback[] callbacks) throws IOException,
          UnsupportedCallbackException {
    for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof SAMLCallback) {
            try {
                SAMLCallback samlCallback = (SAMLCallback) callbacks[i];
                if (samlCallback.getConfirmationMethod().equals(
                        samlCallback.SV_ASSERTION_TYPE)) {
                    samlCallback.setAssertionElement(createSVSAMLAssertion());
                    svAssertion = samlCallback.getAssertionElement();
                } else if (samlCallback.getConfirmationMethod().equals(
                        samlCallback.HOK_ASSERTION_TYPE)) {
                    samlCallback.setAssertionElement(createHOKSAMLAssertion());
                    hokAssertion = samlCallback.getAssertionElement();
                } else {
                    throw new Exception("SAML Assertion Type is not matched.");
                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        } else {
            throw unsupported;
        }
     }
  }

  private static Element createSVSAMLAssertion() {...}
  private static Element createSVSAMLAssertion20() {...}
  private Element createHOKSAMLAssertion() {...}
  private Element createHOKSAMLAssertion20() {...}
  ...
}
```

17.4.5. Secure Conversation

This section describes how to configure secure conversation in server and client applications.

17.4.5.1. Server Configurations

The server-side web service is configured through the service-config.xml web service configuration file.

Configuring Server-Side Secure Conversation: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
        <endpoint-policy-subject>
            <portcomponent-wsdl-name>AddNumbersPort</portcomponent-wsdl-name>
            <addressing-policy>
                <www-w3-org />
            </addressing-policy>
            <security-policy>
                <security-binding>
                    <symmetric-binding>
                        <protection-token>
                            <secure-conversation-token>
                                <asymmetric-binding-initiator-token>
                                    <include-token>true</include-token>
                                </asymmetric-binding-initiator-token>
                                <asymmetric-binding-recipient-token>
                                </asymmetric-binding-recipient-token>
                                <include-token>true</include-token>
                            </secure-conversation-token>
                        </protection-token>
                        <encrypt-signature>true</encrypt-signature>
                    </symmetric-binding>
                </security-binding>
                <trust>true</trust>
                <keystore>...</keystore>
                <truststore>...</truststore>
            </security-policy>
            <operation-policy-subject>
                <operation-wsdl-name>addNumbers</operation-wsdl-name>
                <input-message-policy-subject>
                    <security-policy>
                        <protection>
                            <signed-part>...</signed-part>
                            <encrypted-part>...</encrypted-part>
                        </protection>
                    </security-policy>
                </input-message-policy-subject>
            </operation-policy-subject>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

17.4.5.2. Client Configurations

To configure a client-side web service, a keystore (truststore) must also be configured in the wsit-client.xml file.

Configuring Client-Side Secure Conversation: <wsit-client.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
...
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
    <message name="add" />
    <message name="addResponse" />
    <portType name="NewWebService">
        <wsdl:operation name="add">
            <wsdl:input message="tns:add" />
            <wsdl:output message="tns:addResponse" />
        </wsdl:operation>
    </portType>
    <binding name="NewWebServicePortBinding" type="tns:NewWebService">
        <wsp:PolicyReference URI="#NewWebServicePortBindingPolicy" />
        <soap12:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="add">
            <soap12:operation soapAction="http://xmlsoap.org/Ping"
                style="document" />
            <wsdl:input>
                <soap12:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap12:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </binding>
    <service name="NewWebServiceService">
        <wsdl:port name="NewWebServicePort" binding="tns:NewWebServicePortBinding">
            <soap12:address location="REPLACE_WITH_ACTUAL_ADDRESS" />
        </wsdl:port>
    </service>
    <wsp:Policy wsu:Id="NewWebServicePortBindingPolicy"
    xmlns:scc="http://schemas.sun.com/ws/2006/05/sc/client">
        <wsp:ExactlyOne>
            <wsp:All>
                <sc:KeyStore wspp:visibility="private"
                    location="./keystore.jks" type="JKS"
                    alias="xws-security-client" storepass="changeit">
                </sc:KeyStore>
                <sc:TrustStore wspp:visibility="private"
                    location="./cacerts.jks" type="JKS"
                    storepass="changeit" peeralias="xws-security-server"
                    stsalias="wssip">
                </sc:TrustStore>
                <scc:SCClientConfiguration wspp:visibility="private">
                    <scc:LifeTime>36000</scc:LifeTime>
                </scc:SCClientConfiguration>
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>
</definitions>
```

17.4.6. Web Service Trust

This section describes how to configure a server, STS, and client for WS-Trust.

17.4.6.1. Server Configurations

The server-side web service is configured through the service-config.xml web service configuration file.

Configuring Server Web Service Trust: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
        <endpoint-policy-subject>
<!--
Web service trust runs on the WS-Addressing framework.
-->
            <addressing-policy>
                <www-w3-org />
            </addressing-policy>
            <security-policy>
<!--
Specify that the Web service will use the x509 token as security token.
-->
                <security-binding>
                    <symmetric-binding>
                        <protection-token>
                            <x509-token />
                        </protection-token>
                        <layout>Strict</layout>
                    </symmetric-binding>
                </security-binding>
<!--
Specify that an additional security token for the Web service should be issued by STS of the
following.
-->
                <token>
                    <endorsing-supporting-token>
                        <issued-token>
                            <issuer-address>
                             http://localhost:8088/trust_sts/SecurityTokenService
                            </issuer-address>
                        </issued-token>
                    </endorsing-supporting-token>
                </token>
                <wss-version>11</wss-version>
                <trust>true</trust>
                ...
            </security-policy>
            ...
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

17.4.6.2. STS Configurations

For STS configuration, the WS-SecurityPolicy must be configured through the WS-Policy framework of WSDL.

The following is an example of configuring the WS-SecurityPolicy.

Configuring WS-Security Policy: <sts.wsdl>

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ...>
    <wsp:Policy wsu:Id="TmaxSTSServerPolicy">
        <wsp:ExactlyOne>
            <wsp:All>
                <sp:SymmetricBinding
                xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
                    <wsp:Policy>
                        <sp:ProtectionToken>
                            <wsp:Policy>
                                <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never">
                                    <wsp:Policy>
                                        <sp:RequireDerivedKeys />
                                        <sp:RequireThumbprintReference />
                                        <sp:WssX509V3Token10 />
                                    </wsp:Policy>
                                </sp:X509Token>
                            </wsp:Policy>
                        </sp:ProtectionToken>
                        <sp:AlgorithmSuite>
                            <wsp:Policy>
                                <sp:Basic128 />
                            </wsp:Policy>
                        </sp:AlgorithmSuite>
                        <sp:Layout>
                            <wsp:Policy>
                                <sp:Lax />
                            </wsp:Policy>
                        </sp:Layout>
                        <sp:IncludeTimestamp />
                        <sp:EncryptSignature />
                        <sp:OnlySignEntireHeadersAndBody />
                    </wsp:Policy>
                </sp:SymmetricBinding>
                <sp:SignedSupportingTokens
                xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
                    <wsp:Policy>
                        <sp:UsernameToken
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipient"
>
                            <wsp:Policy>
                                <sp:WssUsernameToken10 />
                            </wsp:Policy>
                        </sp:UsernameToken>
                    </wsp:Policy>
                </sp:SignedSupportingTokens>
                <sp:Wss11
                xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
                    <wsp:Policy>
                        <sp:MustSupportRefKeyIdentifier />
                        <sp:MustSupportRefIssuerSerial />
                        <sp:MustSupportRefThumbprint />
                        <sp:MustSupportRefEncryptedKey />
                    </wsp:Policy>
                </sp:Wss11>
                <sp:Trust10
                xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
                    <wsp:Policy>
                        <sp:MustSupportIssuedTokens />
                        <sp:RequireClientEntropy />
                        <sp:RequireServerEntropy />
                    </wsp:Policy>
                </sp:Trust10>
                ...
                <sc:ValidatorConfiguration
                    xmlns:sc="http://schemas.sun.com/2006/03/wss/server">
                    <sc:Validator name="usernameValidator"
                        classname="trust.sts.UsernamePasswordValidator" />
                </sc:ValidatorConfiguration>
                <tc:STSConfiguration
                    xmlns:tc="http://schemas.sun.com/ws/2006/05/trust/server"
                    encryptIssuedKey="true" encryptIssuedToken="false">
                    <tc:LifeTime>36000</tc:LifeTime>
                    <tc:Contract>
                    com.sun.xml.ws.security.trust.impl.IssueSamlTokenContractImpl
                    </tc:Contract>
                    <tc:Issuer>TmaxsoftSTS</tc:Issuer>
                    <tc:ServiceProviders>

<tc:ServiceProviderendPoint="http://localhost:8088/trust_server/FinancialService">
                            <tc:CertAlias>bob</tc:CertAlias>
```

```
                              <tc:TokenType>
                                http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV1.1
                              </tc:TokenType>
                         </tc:ServiceProvider>
                         <tc:ServiceProvider endPoint="default">
                              <tc:CertAlias>bob</tc:CertAlias>
                              <tc:TokenType>
                                http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV1.1
                              </tc:TokenType>
                         </tc:ServiceProvider>
                      </tc:ServiceProviders>
                  </tc:STSConfiguration>
                  <wsap10:UsingAddressing />
              </wsp:All>
          </wsp:ExactlyOne>
      </wsp:Policy>
      ...
      <wsdl:binding name="ISecurityTokenService_Binding"
          type="tns:ISecurityTokenService">
          <wsp:PolicyReference URI="#TmaxSTSServerPolicy" />
          ...
      </wsdl:binding>
      ...
</wsdl:definitions>
```

17.4.6.3. Client Configurations

To configure a client-side web service, a keystore (truststore) for each server and STS must be configured in the wsit-client.xml file.

Configuring Client Web Service Trust: <wsit-client.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://tempuri.org/" ...>
    <!-- FinancialService WSDL -->
    <wsp:Policy wsu:Id="TmaxFSClientPolicy" ...>
        <wsp:ExactlyOne>
            <wsp:All>
                <sc:KeyStore ... />
                <sc:TrustStore  ... />
                <scc:SCClientConfiguration wspp:visibility="private">
                    <scc:LifeTime>36000</scc:LifeTime>
                </scc:SCClientConfiguration>
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>
    ...
    <wsdl:binding name="IFinancialService_Binding"
        type="tns:IFinancialService">
        <wsp:PolicyReference URI="#TmaxFSClientPolicy" />
        ...
    </wsdl:binding>
    <wsdl:service name="FinancialService">
        <wsdl:port name="IFinancialService_Port"
            binding="tns:IFinancialService_Binding">
            <soap:address
                location="http://localhost:8088/trust_server/FinancialService" />
        </wsdl:port>
    </wsdl:service>

    <!-- STSService WSDL -->
    <wsp:Policy wsu:Id="TmaxSTSClientPolicy" ...>
        <wsp:ExactlyOne>
            <wsp:All>
                <sc:KeyStore ... />
                <sc:TrustStore ... />
                <sc:CallbackHandlerConfiguration
                    xmlns:sc="http://schemas.sun.com/2006/03/wss/client">
                    <sc:CallbackHandler default="alice" name="usernameHandler" />
                    <sc:CallbackHandler default="alice" name="passwordHandler" />
                </sc:CallbackHandlerConfiguration>
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>
    ...
    <wsdl:binding name="ISecurityTokenService_Binding"
        type="tns:ISecurityTokenService">
        <wsp:PolicyReference URI="#TmaxSTSClientPolicy" />
        ...
    </wsdl:binding>
    <wsdl:service name="SecurityTokenService">
        <wsdl:port name="ISecurityTokenService_Port"
            binding="tns:ISecurityTokenService_Binding">
            <soap:address
                location="http://localhost:8088/trust_sts/STSImplService" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

17.4.7. Client Execution

The following is an example of executing a client.

```
<<__Exception__>>
com.sun.xml.ws.server.UnsupportedMediaException: Unsupported Content-Type:
 application/soap+xml Supported ones are: [text/xml]
  at com.sun.xml.ws.encoding.StreamSOAPCodec.decode(StreamSOAPCodec.java:295)
  at com.sun.xml.ws.encoding.StreamSOAPCodec.decode(StreamSOAPCodec.java:129)
  at com.sun.xml.ws.encoding.SOAPBindingCodec.decode(SOAPBindingCodec.java:287)
  at jeus.webservices.jaxws.transport.http.HttpAdapter.decodePacket
(HttpAdapter.java:322)
  at jeus.webservices.jaxws.transport.http.HttpAdapter.access$3
(HttpAdapter.java:287)
  at jeus.webservices.jaxws.transport.http.HttpAdapter$HttpToolkit.handle
(HttpAdapter.java:517)
  at jeus.webservices.jaxws.transport.http.HttpAdapter.handle
(HttpAdapter.java:272)
  at jeus.webservices.jaxws.transport.http.EndpointAdapter.handle
(EndpointAdapter.java:149)
  at jeus.webservices.jaxws.transport.http.servlet.WSServletDelegate.doGet
(WSServletDelegate.java:139)
```

```
  at jeus.webservices.jaxws.transport.http.servlet.WSServletDelegate.doPost
(WSServletDelegate.java:170)
  at jeus.webservices.jaxws.transport.http.servlet.WSServlet.doPost
(WSServlet.java:23)
  at jakarta.servlet.http.HttpServlet.service(HttpServlet.java:725)
  at jakarta.servlet.http.HttpServlet.service(HttpServlet.java:818)
  at jeus.servlet.engine.ServletWrapper.executeServlet(ServletWrapper.java:328)
  at jeus.servlet.engine.ServletWrapper.execute(ServletWrapper.java:222)
  at jeus.servlet.engine.HttpRequestProcessor.run(HttpRequestProcessor.java:278)
<<__!Exception__>>
[2999.01.01 00:00:00][0][bxxx] [TMAX-xx] Unsupported Content-Type:
 application/soap+xml Supported ones are: [text/xml]
<<__Exception__>>
com.sun.xml.ws.server.UnsupportedMediaException: Unsupported Content-Type:
 application/soap+xml Supported ones are: [text/xml]
  at com.sun.xml.ws.encoding.StreamSOAPCodec.decode(StreamSOAPCodec.java:295)
  at com.sun.xml.ws.encoding.StreamSOAPCodec.decode(StreamSOAPCodec.java:129)
  at com.sun.xml.ws.encoding.SOAPBindingCodec.decode(SOAPBindingCodec.java:287)
  at jeus.webservices.jaxws.transport.http.HttpAdapter.decodePacket
(HttpAdapter.java:322)
  at jeus.webservices.jaxws.transport.http.HttpAdapter.access$3
(HttpAdapter.java:287)
  at jeus.webservices.jaxws.transport.http.HttpAdapter$HttpToolkit.handle
(HttpAdapter.java:517)
  at jeus.webservices.jaxws.transport.http.HttpAdapter.handle(HttpAdapter.java:272)
  at jeus.webservices.jaxws.transport.http.EndpointAdapter.handle
(EndpointAdapter.java:149)
  at jeus.webservices.jaxws.transport.http.servlet.WSServletDelegate.doGet
(WSServletDelegate.java:139)
  at jeus.webservices.jaxws.transport.http.servlet.WSServletDelegate.doPost
(WSServletDelegate.java:170)
  at jeus.webservices.jaxws.transport.http.servlet.WSServlet.doPost
(WSServlet.java:23)
  at jakarta.servlet.http.HttpServlet.service(HttpServlet.java:725)
  at jakarta.servlet.http.HttpServlet.service(HttpServlet.java:818)
  at jeus.servlet.engine.ServletWrapper.executeServlet(ServletWrapper.java:328)
  at jeus.servlet.engine.ServletWrapper.execute(ServletWrapper.java:222)
  at jeus.servlet.engine.HttpRequestProcessor.run(HttpRequestProcessor.java:278)
<<__!Exception__>>
user alice is a validate user.
your company : Tmaxsoft
your department : Infra
```

Once the client is executed, the client attempts to connect through various URL suffixes in order to get the STS URL information. If the log level is set to a value higher than 'FINE', the exception in the previous example will occur.

17.5. How to Migrate JAX-RPC (JEUS 5) Web Service Security

As mentioned already, the message-level security in JEUS web services works through the web service security policy. The web service security policy can be directly set in the WSDL document or wsit-endpoint.xml file.

The web service policy can be configured in the service-config.xml file, which supports the web service security policy. If the service-config.xml file can be shared between the server and client, any user can easily process message-level security on both sides by using the file, without understanding the web service and web service security policies. This is similar to the message-level security for the JAX-RPC (JEUS 5) web service.

17.5.1. Encryption

The encryption settings for the message-level security in web services are divided into the following.

- Configuring JAX-RPC web services
- Configuring JAX-WS web services

Configuring JAX-RPC Web Services

The encryption settings for the message-level security in JAX-RPC web services are configured in the jeuswebservices- dd.xml file (servers) or the jeus-web-dd.xml file (client), as shown in the following example.

Configuring JAX-RPC Web Service Encryption: <jeus-webservices-dd.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jeus-webservices-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <service>
        <webservice-description-name>
            PingSecurityService
        </webservice-description-name>
        <port>
            <port-component-name>PingPort</port-component-name>
            <security>
                <request-receiver>
                    <action-list>Encrypt</action-list>
                    <password-callback-class>
                        ping.PingPWCallback
                    </password-callback-class>
                    <decryption>
                        <keystore>
                            <key-type>jks</key-type>
                            <keystore-password>changeit</keystore-password>
                            <keystore-filename>
                                server-keystore.jks
                            </keystore-filename>
                        </keystore>
                    </decryption>
                </request-receiver>
                <response-sender>
                    <action-list>Encrypt</action-list>
                    <password-callback-class>
                        ping.PingPWCallback
                    </password-callback-class>
                    <encryption-infos>
                        <encryption-info>
                            <encryptionUser>xws-security-client</encryptionUser>
```

```
                              <keyIdentifier>DirectReference</keyIdentifier>
                              <keystore>
                                  <key-type>jks</key-type>
                                  <keystore-password>changeit</keystore-password>
                                  <keystore-filename>
                                      server-truststore.jks
                                  </keystore-filename>
                              </keystore>
                          </encryption-info>
                      </encryption-infos>
                  </response-sender>
              </security>
          </port>
      </service>
</jeus-webservices-dd>
```

Configuring JAX-WS Web Services

The message-level security in JAX-WS (JEUS 9) web services is configured by including the 'service-config.xml' as the argument of the '-policy' option when creating a web service by using the wsgen or wsimport tool.

Configuring JAX-WS (JEUS 9) Web Service Encryption: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
        <endpoint-policy-subject>
            <portcomponent-wsdl-name>PingPort</portcomponent-wsdl-name>
            <security-policy>
                <security-binding>
                    <asymmetric-binding>
                        <initiator-token>
                            <x509-token/>
                        </initiator-token>
                        <recipient-token>
                            <x509-token/>
                        </recipient-token>
                    </asymmetric-binding>
                </security-binding>
                <keystore>
                    <keystore-file>
                        <alias>xws-security-client</alias>
                        <key-type>JKS</key-type>
                        <keystore-password>changeit</keystore-password>
                        <keystore-filename>
                            client-keystore.jks
                        </keystore-filename>
                    </keystore-file>
                </keystore>
                <truststore>
                    <keystore-file>
                        <alias>xws-security-server</alias>
                        <key-type>JKS</key-type>
                        <keystore-password>changeit</keystore-password>
                        <keystore-filename>
                            client-truststore.jks
                        </keystore-filename>
                    </keystore-file>
                </truststore>
            </security-policy>
            <operation-policy-subject>
                <operation-wsdl-name>ping</operation-wsdl-name>
                <input-message-policy-subject>
                    <security-policy>
                        <protection>
                            <encrypted-part>
                                <body/>
                            </encrypted-part>
                        </protection>
                    </security-policy>
                </input-message-policy-subject>
                <output-message-policy-subject>
                    <security-policy>
                        <protection>
                            <encrypted-part>
                                <body/>
                            </encrypted-part>
                        </protection>
                    </security-policy>
                </output-message-policy-subject>
            </operation-policy-subject>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

17.5.2. Signature

Signature settings for message-level security is divided into the following.

- Configuring JAX-RPC web services
- Configuring JAX-WS web services

Configuring JAX-RPC Web Services

The signature settings for the message-level security in JAX-RPC web services are configured in the jeus-webservices- dd.xml file (server) or the jeus-web-dd.xml file (client), as shown in the following.

Configuring JAX-RPC Web Service Signature: <jeus-webservices-dd.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jeus-webservices-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <service>
        <webservice-description-name>
            PingSecurityService
        </webservice-description-name>
        <port>
            <port-component-name>PingPort</port-component-name>
            <security>
                <request-receiver>
```

```
                    <action-list>Signature</action-list>
                    <password-callback-class>
                        ping.PingPWCallback
                    </password-callback-class>
                    <signature-verification>
                        <keystore>
                            <key-type>jks</key-type>
                            <keystore-password>changeit</keystore-password>
                            <keystore-filename>
                                server-truststore.jks
                            </keystore-filename>
                        </keystore>
                    </signature-verification>
                </request-receiver>
                <response-sender>
                    <action-list>Signature</action-list>
                    <password-callback-class>
                        ping.PingPWCallback
                    </password-callback-class>
                    <user>xws-security-server</user>
                    <signature-infos>
                        <signature-info>
                            <keyIdentifier>DirectReference</keyIdentifier>
                            <keystore>
                                <key-type>jks</key-type>
                                <keystore-password>changeit</keystore-password>
                                <keystore-filename>
                                    server-keystore.jks
                                </keystore-filename>
                            </keystore>
                        </signature-info>
                    </signature-infos>
                </response-sender>
            </security>
        </port>
    </service>
</jeus-webservices-dd>
```

Configuring JAX-WS Web Services

The message-level security in JAX-WS (JEUS 9) web services is configured by including the 'service-config.xml' as the argument of the '-policy' option when creating a web service by using the wsgen or wsimport tool.

Configuring JAX-WS(JEUS 9) Web Service Signature: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
        <endpoint-policy-subject>
            <portcomponent-wsdl-name>PingPort</portcomponent-wsdl-name>
            <security-policy>
                <security-binding>
                    <asymmetric-binding>
                        <initiator-token>
                            <x509-token/>
                        </initiator-token>
                        <recipient-token>
                            <x509-token/>
                        </recipient-token>
                    </asymmetric-binding>
                </security-binding>
                <keystore>
                    <keystore-file>
                        <alias>xws-security-client</alias>
                        <key-type>JKS</key-type>
                        <keystore-password>changeit</keystore-password>
                        <keystore-filename>
                            client-keystore.jks
                        </keystore-filename>
                    </keystore-file>
                </keystore>
                <truststore>
                    <keystore-file>
                        <alias>xws-security-server</alias>
                        <key-type>JKS</key-type>
                        <keystore-password>changeit</keystore-password>
                        <keystore-filename>
                            client-truststore.jks
                        </keystore-filename>
                    </keystore-file>
                </truststore>
            </security-policy>
            <operation-policy-subject>
                <operation-wsdl-name>ping</operation-wsdl-name>
                <input-message-policy-subject>
                    <security-policy>
                        <protection>
                            <signed-part>
                                <body/>
                            </signed-part>
                        </protection>
                    </security-policy>
                </input-message-policy-subject>
                <output-message-policy-subject>
                    <security-policy>
                        <protection>
                            <signed-part>
                                <body/>
                            </signed-part>
                        </protection>
                    </security-policy>
                </output-message-policy-subject>
            </operation-policy-subject>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

17.5.3. Timestamp

Timestamp settings for message-level security is divided into the following.

Configuring JAX-WS web services

The message-level security in JAX-WS (JEUS 9) web services is configured by including the 'service-config.xml' as the argument of the '-policy' option when creating a web service by using the wsgen or wsimport tool.

Configuring JAX-WS (JEUS 9) Web Service Timestamp: <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
        <endpoint-policy-subject>
            <portcomponent-wsdl-name>PingPort</portcomponent-wsdl-name>
            <security-policy>
                <security-binding>
                    <asymmetric-binding>
                        <initiator-token>
                            <x509-token/>
                        </initiator-token>
                        <recipient-token>
                            <x509-token/>
                        </recipient-token>
                    </asymmetric-binding>
                    <timestamp>true</timestamp>
                </security-binding>
                <keystore>
                    <keystore-file>
                        <alias>xws-security-client</alias>
                        <key-type>JKS</key-type>
                        <keystore-password>changeit</keystore-password>
                        <keystore-filename>
                            client-keystore.jks
                        </keystore-filename>
                    </keystore-file>
                </keystore>
                <truststore>
                    <keystore-file>
                        <alias>xws-security-server</alias>
                        <key-type>JKS</key-type>
                        <keystore-password>changeit</keystore-password>
                        <keystore-filename>
                            client-truststore.jks
                        </keystore-filename>
                    </keystore-file>
                </truststore>
            </security-policy>
            <operation-policy-subject>
                <operation-wsdl-name>ping</operation-wsdl-name>
                <input-message-policy-subject>
                    <security-policy>
                        <protection>
                            <signed-part>
                                <body/>
                            </signed-part>
                        </protection>
                    </security-policy>
                </input-message-policy-subject>
                <output-message-policy-subject>
                    <security-policy>
```

```
                        <protection>
                            <signed-part>
                                <body/>
                            </signed-part>
                        </protection>
                    </security-policy>
                </output-message-policy-subject>
            </operation-policy-subject>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

17.5.4. Username Token

Username token settings for message-level security is divided into the following.

Configuring JAX-WS web services

The message-level security in JAX-WS (JEUS 9) web services is configured by including the 'service-config.xml' as the argument of the '-policy' option when creating a web service by using the wsgen or wsimport tool.

Configuring JAX-WS (JEUS 9) Web Service Username Token : <service-config.xml>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<web-services-config xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <policy>
        <endpoint-policy-subject>
            <security-policy>
                <security-binding>
                    <symmetric-binding>
                        <protection-token>
                            <x509-token>
                                <include-token>true</include-token>
                            </x509-token>
                        </protection-token>
                    </symmetric-binding>
                </security-binding>
<token>
                    <signed-supporting-token>
                        <username-token>
                            <username-password-validator>
                                fromjava.server.UsernamePasswordValidator
                            </username-password-validator>
                            <include-token>true</include-token>
                        </username-token>
                    </signed-supporting-token>
                </token>
<keystore>
                    <keystore-file>
                        <alias>xws-security-server</alias>
                        <key-type>JKS</key-type>
                        <keystore-password>changeit</keystore-password>
                        <keystore-filename>
                            keystore.jks
                        </keystore-filename>
                    </keystore-file>
                </keystore>
                <wss-version>11</wss-version>
                <protection>
                    <signed-part>
                        <body/>
                    </signed-part>
                    <encrypted-part>
                        <body/>
                    </encrypted-part>
                </protection>
            </security-policy>
        </endpoint-policy-subject>
    </policy>
</web-services-config>
```

17.6. How to Invoke Access-Controlled Web Services

Access control can be set for a JEUS web service to only allow users with permissions to invoke the web service, and the setting is different for each web service back-end. Access control can be set in the same way as described in the Configuring Access Control section.

Portable artifacts must be created based on the information in WSDL in order to invoke a web service with access control settings (basic authentication). If a username and password is required for the service, they must also be configured.

17.6.1. Creating Portable Artifacts

Portable artifacts must be created for the clients of an access-controlled web service.

Enter the following command in the console to create portable artifacts by using the **wsimport** tool.

```
$ wsimport -Xauthfile authorization.txt
http://localhost:8088/AddNumbers/AddNumbersImplService?wsdl
```

The "authorization.txt" file is an authorization configuration file required for accessing the WSDL. Refer to the following example.

Configuring Authorization: <authorization.txt>

```
http://jeus:jeus@localhost:8088/AddNumbers/AddNumbersImplService?wsdl
```

In the previous example, the user name and password are both 'jeus'.

17.6.2. Creating Web Service Clients

The following settings must be inserted into the client program in order for JAX-WS web service client to authenticate itself.

- jakarta.xml.ws.BindingProvider.USERNAME_PROPERTY
- jakarta.xml.ws.BindingProvider.PASSWORD_PROPERTY

The following example shows how the aforementioned settings can be inserted into the client program.

```
Echo port = // … SEI(Service Endpoint Interface)의 획득
((jakarta.xml.ws.BindingProvider) port).getRequestContext().
    put(jakarta.xml.ws.BindingProvider.USERNAME_PROPERTY, "jeus");
((jakarta.xml.ws.BindingProvider) port).getRequestContext().
    put(jakarta.xml.ws.BindingProvider.PASSWORD_PROPERTY, "jeus");
String s = port.echoString("JEUS");
```

At runtime, the JAX-WS web service client engine accesses the remote WSDL to create a dynamic stub, which requires additional authorization. Set the authorization for the client program by using the java.net.Authenticator class.

The following is an example of a client program that uses the java.net.Authenticator class.

Web Service Client : <AddNumberClient.java>

```
public class AddNumbersClient {
  ...
  public void execute() {
      Authenticator.setDefault(new MyAuthenticator());

      AddNumbersImpl port = new AddNumbersImplService().getAddNumbersImplPort();
      ((jakarta.xml.ws.BindingProvider) port).getRequestContext().
            put(jakarta.xml.ws.BindingProvider.USERNAME_PROPERTY, "jeus");
      ((jakarta.xml.ws.BindingProvider) port).getRequestContext().
            put(jakarta.xml.ws.BindingProvider.PASSWORD_PROPERTY, "jeus");

      int number1 = 10;
      int number2 = 20;

      port.addNumbers(number1, number2);
      ...
  }

  class MyAuthenticator extends Authenticator {
      protected PasswordAuthentication getPasswordAuthentication() {
          return new PasswordAuthentication("jeus", "jeus".toCharArray());
      }
  }
}
```

17.7. Configuring Access Control

Access control configuration can be set to only allow authorized users to call the web service. The access control configuration varies depending on the web service backend. This section describes how to set the web service access control, and how to call an access-controlled web service.

17.7.1. Configuring Access Control Security for Java Web Services

Web service access can be controlled by applying access control to the URL of the web service. In order to apply access control, security information has to be specified in the deployment descriptor files ('web.xml' and 'jeus-web-dd.xml'). A secure domain configuration is also required.

Configuring Secure Domain

First, a user must be registered to apply access control.

Register the user in the following file.

```
JEUS_HOME/config/{NODE_NAME}/security/{SECURITY_DOMAIN_NAME}/accounts.xml
```

The following is an example of registering a user with the username of "jeus" and password of "jeus". The password must be encrypted using Base64 encoding.

Configuring Access Control: <accounts.xml>

```
<accounts xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <users>
        <user>
            <name>jeus</name>
            <password>{SHA}McbQlyhI3yiOG1HGTg8DQVWkyhg=</password>
        </user>
    </users>
</accounts>
```

Configuring a DD file

Add the following <role-mapping> element to the jeus-web-dd.xml file.

Configuring Access Control: <jeus-web-dd.xml>

```
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <docbase>BA_DocLitEchoService</docbase>
    <role-mapping>
        <role-permission>
            <principal>jeus</principal>
            <role>Administrator</role>
        </role-permission>
    </role-mapping>
```

```
</jeus-web-dd>
```

Then add the following security related settings to the web.xml file.

Configuring Access Control: <web.xml>

```
<web-app…>
    . . .
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>MySecureBit</web-resource-name>
            <url-pattern>/BA_DocLitEchoService</url-pattern>
            <http-method>POST</http-method>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>Administrator</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>NONE</transport-guarantee>
        </user-data-constraint>
    </security-constraint>

    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>default</realm-name>
    </login-config>
</web-app>
```

When the previous setting is applied, only the user with the username "jeus" and password "jeus" can access the URL, '/BA_DocLitEchoService'.

17.7.2. Configuring Access Control for EJB Web Services

Similar to a Java web service, the access to an EJB web service can be controlled by applying access control to the web service URL. In order to apply access control, security information has to be specified in the deployment descriptor files ('ejb-jar.xml' and 'jeus-ejb-dd.xml') and the web service deployment descriptor file ('jeus-webservices-dd.xml'). A secure domain configuration is also required.

Configuring Secure Domain

First, a user must be registered to apply access control.

Register the user in the following file.

```
JEUS_HOME/config/{NODE_NAME}/security/{SECURITY_DOMAIN_NAME}/accounts.xml
```

The following is an example of registering a user with the username of "jeus" and password of "jeus".

The password must be encrypted using Base64 encoding.

Access Control for EJB Web Services: <accounts.xml>

```
<accounts xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <users>
        <user>
            <name>jeus</name>
            <password>{SHA}McbQlyhI3yiOG1HGTg8DQVWkyhg=</password>
        </user>
    </users>
</accounts>
```

Configuring a DD file

Add the following <role-permission> element to the jeus-ejb-dd.xml file.

Access Control for EJB Web Services: <jeus-ejb-dd.xml>

```
<jeus-ejb-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <module-info>
        <role-permission>
            <principal>jeus</principal>
            <role>Administrator</role>
        </role-permission>
    </module-info>
    <beanlist>
        ...
    </beanlist>
</jeus-ejb-dd>
```

Then add the following security related settings to the ejb-jar.xml file.

Access Control for EJB Web Services: <ejb-jar.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar ...>
    <display-name>AddressEjb</display-name>
    <enterprise-beans>
        ...
    </enterprise-beans>
    <assembly-descriptor>
        <security-role>
            <role-name>Administrator</role-name>
        </security-role>
        <method-permission>
            <role-name>Administrator</role-name>
            <method>
                <ejb-name>AddressEJB</ejb-name>
                <method-intf>ServiceEndpoint</method-intf>
                <method-name>listAll</method-name>
            </method>
        </method-permission>
    </assembly-descriptor>
```

```
</ejb-jar>
```

The following configurations can be additionally set in the web service deployment descriptor. Default values are used if they are not configured. In order to guarantee the data integrity of SSL socket communication, the <ejb-transport-guarantee> must be set to 'INTEGRAL'. If both the confidentiality and data integrity must be guaranteed, set it to 'CONFIDENTIAL'. A separate HTTP listener configuration is also required to enable HTTPS communication.

Access Control for EJB Web Services: <jeus-webservices-dd.xml>

```
<jeus-webservices-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <ejb-context-path>webservice</ejb-context-path>
<ejb-login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>default</realm-name>
    </ejb-login-config>
    <service>
        <webservice-description-name>…</webservice-description-name>
        <port>
            <port-component-name>…</port-component-name>
            <ejb-endpoint-url>… </ejb-endpoint-url>
<ejb-transport-guarantee>
                CONFIDENTIAL
            </ejb-transport-guarantee>
        </port>
    </service>
</jeus-webservices-dd>
```

17.7.3. Invoking Web Services Configured with Basic Authentication

In order to invoke a web service that is configured with basic authentication, a stub must be generated based on the information from the WSDL. The stub is used to call the web service, and, if a username and password are required, they must also be configured.

Creating a Stub from WSDL

When using a console tool, use the following options.

```
wsdl2java -gen:client -username jeus -password jeus
http://localhost:8088/BA_DocLitEchoService/BA_DocLitEchoService?wsdl
```

18. Server-Sent Event

This chapter describes how to use HTML5's Server-Sent Event (SSE) in JEUS.

18.1. Overview

Server-Sent Events enables servers to push data to webpages by using the standard HTTP or HTTPS via one-way client server connection. In the Server-Sent Events communication model, the client (e.g. browser) maintains the initial connection, and the server provides data and sends it to the client.

The Server-Sent Events technology is part of the HTML5 specification, and is supported in JEUS through JAX-RS (Java API for RESTful Web Services) 2.0.

18.2. Server-Sent Events (SSE) Support in the JAX-RS Resource

Add SSeFeature in the resource for SSE support.

Simple SSE Resource Method

```
...
import jeus.webservices.jaxrs.media.sse.EventOutput;
import jeus.webservices.jaxrs.media.sse.OutboundEvent;
import jeus.webservices.jaxrs.media.sse.SseFeature;
...

@Path("events")
public class SseResource {

    @GET
    @Produces(SseFeature.SERVER_SENT_EVENTS)
    public EventOutput getServerSentEvents() {
        EventOutput eventOutput = new EventOutput();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    for (int i = 0; i < 5; i++) {
                        // ... code that waits a few seconds
                        OutboundEvent.Builder eventBuilder = new OutboundEvent.Builder();
                        eventBuilder.name("example");
                        eventBuilder.data(String.class, "Hello world " + i + "!");
                        OutboundEvent event = eventBuilder.build();
                        eventOutput.write(event);
                    }
                } catch (IOException e) {
                    throw new RuntimeException(e);
                } finally {
                    try {
                        eventOutput.close();
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                }
            }
        }).start();
        return eventOutput;
    }
}
```

The above source code defines the resource deployed to URI "/events". The resource contains a @GET resource method which returns the EventOutput entity for processing ChunkedOutput messages.

After EventOutput is returned from the method, by recognizing EventOutput as ChunkedOutput, the JAX-RS runtime does not immediately disconnect from the client. It writes an HTTP header in the response script and waits for the sent chunks (SSE events). Then, the client reads the header and starts listening to individual events.

In the previous example, a resource method creates a thread for sending a series of five events. There is a small delay between the series of events. Each event is expressed as an OutboundEvent type. An OutboundEvent is applied in the standard SSE message format, and includes a property that expresses a name, comment, and id. The dat (Class) method is used to serialize event data.

A client connected to an SSE supported resource will receive the following data from the entity stream.

```
event: example
data: Hello world 0!

event: example
data: Hello world 1!

event: example
data: Hello world 2!

event: example
data: Hello world 3!

event: example
data: Hello world 4!
```

18.3. SSE Event Processing in the JAX-RS Client

JEUS JAX-RS uses two types of program models to provide a client API for receiving and handling SSE events.

- Pull model: Pools events from EventInput
- Push model: Listens EventSource’s asynchronous notifications

18.3.1. SSE Event Reading with EventInput

A client can read an event from EventInput. Let's take a look at the following code.

SSE Event Reading with EventInput

```
Client client = ClientBuilder.newBuilder().build();
WebTarget target = client.target("http://localhost:8088/sse_example/events");
EventInput eventInput = target.request().get(EventInput.class);
while (!eventInput.isClosed()) {
    final InboundEvent inboundEvent = eventInput.read();
    if (inboundEvent == null) {
        // connection has been closed
        break;
    }
    System.out.println(inboundEvent.getName() + ": " + inboundEvent.readData(String.class));
}
```

A client connects to the server where the SseResource in Simple SSE Resource Method is deployed. First create the JAX_RS client object. Then, get the WebTarget object from the client object, and use it for calling an HTTP request. The returned response entity can be read via EventInput. In the example source code, a loop is started in order to read the inbound SSE event from the eventInput response stream. An individual chunk read from eventInput is an InboundEvent. The InboundEvent.readData(Class) is a method used for de-serializing event data.

The following is the result.

```
example: Hello world 0!
example: Hello world 1!
example: Hello world 2!
example: Hello world 3!
example: Hello world 4!
```

18.3.2. Asynchronous SSE Processing by using EventSource

EventSource is a client API used to read SSE events asynchronously. The following example describes how to use EventSource.

Reading an SSE Event with EventSource

```
Client client = ClientBuilder.newBuilder().build();
WebTarget target = client.target("http://localhost:8088/sse_example/events");
EventSource eventSource = new EventSource(target);
EventListener listener = new EventListener() {
        @Override
        public void onEvent(InboundEvent inboundEvent) {
            System.out.println(inboundEvent.getName() + ": " + inboundEvent.readData(String.class));
        }
    };
eventSource.addEventListener("example", listener);
eventSource.open();
...
```

```
eventSource.close();
```

The client connects to the server where the SseResource in Simple SSE Resource Method example is deployed. First, create the JAX-RS client object. Then, get the WebTarget object from the client object. HTTP request calls for a web target is not created in the WebTarget object. Create an EventSource object by initializing it as a target object. A created EventSource object does not automatically connect to the target. To connect, manually connect by using the eventSource.open() method.

The EventListener implementation is used for listening and handling the incoming SSE event. The InboundEvent.readData(Class) method is used for de-serializing event data from an inboundEvent.

The listener shows the following result in an example.

```
example: Hello world 0!
example: Hello world 1!
example: Hello world 2!
example: Hello world 3!
example: Hello world 4!
```

19. XML of JEUS Web Services

This chapter describes various XML related technologies provided by JEUS 9.

19.1. Overview

This chapter will discuss about Java Architecture for XML Binding (JAXB), which enables the XML document information to be handled programmatically by binding the XML document to Java classes that are compiled from the schema. Additionally, Streaming APIs For XML (StAX), for which APIs are currently being added to JAXP, will also be discussed.

- JAXB(Java Architecture for XML Binding)
- JAXP(Java API for XML Processing)
- SJSXP(Sun Java Streaming XML Parser)

The following are terms that are commonly used in the context of Java object conversion and XML documents.

Term	Description
Unmarshalling	Process of converting XML document or XML content into objects(Java content tree) by using Java class
Marshalling	Process of converting a Java object (Java content tree) into an XML document or XML content

19.2. Java Architecture for XML Binding (JAXB)

An application related to an XML document that follows a schema must be able to create, modify, or read the document. Data binding for XML used in such application can be performed by using the SAX or DOM API, but keeping up with schema changes is not trivial. Thus, JAXB provides API and tools to automatically map XML documents to Java objects.

The following are features of JAXB.

- Unmarshalling, which converts XML contents into Java object
- Accessing or modifying Java objects
- Marshalling, which converts Java objects into XML contents

As previously stated, JAXB provides a developer with standardized and effective mapping between XML and Java codes. JAXB makes it easier to develop applications that use XML and web service technologies. The following sections will discuss the features of JAXB as well as the tools provided by JAXB in further detail with some example programs.

19.2.1. Binding Compiler (XJC) Related Programming Techniques

Java classes must already exist before converting XML contents into Java objects. The Java classes are created from the schema, and this process is called binding compilation.

This section discusses about XJC, the default binding compiler tool provided by JEUS 9 web services.

XJC

Execute the following command from the command line.

```
JEUS_HOME/bin$ xjc.cmd -help
```

The following describes how to use the tool.

```
Usage: xjc [-options ...] <schema file/URL/dir/jar> ...
           [-b <bindinfo>] ...
If dir is specified, all schema files in it will be compiled.
If jar is specified, /META-INF/sun-jaxb.episode binding file will be
compiled.
Options:
  -nv                 :  do not perform strict validation of the input
                         schema(s)
  -extension          :  allow vendor extensions - do not strictly
                         follow the Compatibility Rules and App E.2
                         from the JAXB Spec
  -b <file/dir>       :  specify external bindings files (each <file>
                         must have its own -b)
                         If a directory is given, **/*.xjb is searched
  -d <dir>            :  generated files will go into this directory
  -p <pkg>            :  specifies the target package
  -httpproxy <proxy> :  set HTTP/HTTPS proxy. Format is
                         [user[:password]@]proxyHost:proxyPort
  -httpproxyfile <f> :  Works like -httpproxy but takes the argument
                         in a file to protect password
  -classpath <arg>    :  specify where to find user class files
  -catalog <file>     :  specify catalog files to resolve external
                         entity references support TR9401, XCatalog,
                         and OASIS XML Catalog format.
  -readOnly           :  generated files will be in read-only mode
  -npa                :  suppress generation of package level
                         annotations(**/package-info.java)
  -no-header          :  suppress generation of a file header with
                         timestamp
  -target 2.0         :  behave like XJC 2.0 and generate code that
                         doesnt use any 2.1 features.
  -xmlschema          :  treat input as W3C XML Schema (default)
  -relaxng            :  treat input as RELAX NG (experimental,
                         unsupported)
  -relaxng-compact    :  treat input as RELAX NG compact syntax
                         (experimental, unsupported)
  -dtd                :  treat input as XML DTD (experimental,
                         unsupported)
  -wsdl               :  treat input as WSDL and compile schemas inside
                         it(experimental,unsupported)
  -verbose          :  be extra verbose
  -quiet            :  suppress compiler output
  -help             :  display this help message
  -version          :  display version information


Extensions:
  -Xlocator         :  enable source location support for generated
                       code
  -Xsync-methods    :  generate accessor methods with the
                       'synchronized' keyword
  -mark-generated   :  mark the generated code as @jakarta.annotation.
                       Generated
  -episode <FILE>   :  generate the episode file for separate
                       compilation
```

JEUS 9 web services also support ANT TASK of XJC. For more information, refer to "xjc" console tool and "xjc" Ant Task in *JEUS Reference Guide*.

Programming Using XJC ANT TASK

The following example shows how to programmatically handle contents of an XML document by converting the schema and XML documents into Java objects in memory through JAXB.

The following describes the overall workflow.

1. Convert an XML document into a Java object by unmarshalling.
2. Programmatically process the converted Java object.
3. Convert the Java object into an XML document via marshalling (output is shown in the example).

The following is an extract from the build.xml file of the example.

Programming with XJC Ant Task: <build.xml>

```
...
<project basedir="." default="run">
...
    <taskdef name="xjc" classname="com.sun.tools.xjc.XJCTask">
        <classpath refid="classpath" />
    </taskdef>
    <target name="compile" description="Compile all Java source
        files">
        <echo message="Compiling the schema..." />
        <mkdir dir="gen-src" />
        <xjc extension="true" schema="po.xsd" package="primer.myPo"
            destdir="gen-src">
            <produces dir="gen-src/primer.myPo" includes="**/*.java" />
        </xjc>
        <echo message="Compiling the java source files..." />
        <mkdir dir="classes" />
        <javac destdir="classes" debug="on">
            <src path="src" />
            <src path="gen-src" />
            <classpath refid="classpath" />
        </javac>
    </target>
    <target name="run" depends="compile" description="Run the sample
        app">
        <echo message="Running the sample application..." />
        <java classname="Main" fork="true">
            <classpath refid="classpath" />
        </java>
    </target>
...
</project>
```

The following describes the 'Main.java' file in the example.

Programming with XJC Ant Task (1): <Main.java>

```
Schema schema = SchemaFactory.newInstance(W3C_XML_SCHEMA_NS_URI).
    newSchema(new File("src/conf/ts.xsd"));

JAXBContext jc = JAXBContext.newInstance("com.tmaxsoft");

//
Unmarshaller unmarshaller = jc.createUnmarshaller();
unmarshaller.setSchema(schema);

...

//
Marshaller marshaller = jc.createMarshaller();
marshaller.setSchema(schema);
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

...
```

Create a JAXBContext object by using the package name of the Java classes, which were created by using the XJC ANT TASK mapping in the build.xml file. Use it to create the Unmarshaller and Marshaller. In addition, create a schema object that determines whether the XML document is implemented according to the schema, and register it with the Unmarshaller and Marshaller.

Programming with XJC Ant Task (2): <Main.java>

```
Object ts = unmarshaller.unmarshal(new File("src/conf/tsInput.xml"));
TmaxSoftType tst = (TmaxSoftType) ((JAXBElement) ts).getValue();
Address address = tst.getAddress1();
address.setName("John Bob");
address.setStreet("242 Main Street");
address.setCity("Beverly Hills");
```

Unmarshal an XML document, called 'poInput.xml' and then modify the unmarshalled Java object. It will be marshalled to the console.

Programming with XJC Ant Task (3) : <Main.java>

```
marshaller.marshal(ts, System.out);
```

19.2.2. Schemagen Programming Techniques

JEUS 9 web services provide a tool to create a particular XML schema from user-implemented Java classes and XJC, a schema compiler that creates a Java class from the schema. The tool is called schema generator, and this section discusses Schemagen, the default schema generator provided by JEUS 9 web services.

Schemagen

Execute the following command from the command line.

```
JEUS_HOME/bin$ schemagen.cmd -help
```

The following describes how to use the tool.

```
Usage: schemagen [-options ...] <java files>
Options:
    -d <path>         :  specify where to place processor and javac
                         generated class files
    -cp <path>        :  specify where to find user specified files
    -classpath <path> :  specify where to find user specified files
    -episode <file>   :  generate episode file for separate
                         compilation
    -version          :  display version information
    -help             :  display this usage message
```

JEUS 9 web services also support ANT TASK for Schemagen. For more information, refer to "schemagen" console tool and "schemagen" Ant Task in *JEUS Reference Guide*.

Programming with Schemagen ANT TASK

The following describes the overall workflow.

1. Create a schema at the source level by processing Java sources with Schemagen.
2. Compile Java sources.
3. Create a Java object by programmatically entering data for the compiled Java sources.
4. Marshal (outputs of the example) the Java object by using the already created schema.

The following is an extract from the build.xml file of the example.

Programming with Schemagen Ant Task: <build.xml>

```
...
<project basedir="." default="run">
...
    <taskdef name="schemagen"
        classname="com.sun.tools.jxc.SchemaGenTask">
        <classpath refid="classpath" />
    </taskdef>
    <target name="compile"
        description="Compile all Java source files">
        <echo message="Generating schemas..." />
        <mkdir dir="schemas" />
        <schemagen destdir="schemas">
            <src path="src" />
            <classpath refid="classpath" />
        </schemagen>
        <echo message="Compiling the java source files..." />
        <mkdir dir="classes" />
        <javac destdir="classes" debug="on">
            <src path="src" />
            <classpath refid="classpath" />
        </javac>
    </target>
    <target name="run" depends="compile"
        description="Run the sample app">
        <echo message="Running the sample application..." />
        <java classname="Main" fork="true">
            <classpath refid="classpath" />
        </java>
    </target>
...
</project>
```

The following are Java classes that represent the same schema in the example.

- BusinessCard.java
- Address.java
- jaxb.index
- package-info.java
- Main.java

The following shows an extract from the 'Main.java' file.

Programming with Schemagen Ant Task (1): <Main.java>

```
BusinessCard card =
    new BusinessCard("John Doe", "Sr. Widget Designer", "Acme, Inc.",
        new Address(null, "123 Widget Way", "Anytown", "MA", (short) 12345),
        "123.456.7890", null, "123.456.7891", "John.Doe@Acme.ORG");

JAXBContext context = JAXBContext.newInstance(BusinessCard.class);
Marshaller m = context.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
m.marshal(card, new FileOutputStream(new File("src/conf/bcard.xml")));
```

```
...
```

Create a Java object by using the compiled Java classes, and register a Marshaller to marshal them to the bcard.xml file.

Programming with Schemagen Ant Task (2): <Main.java>

```
Unmarshaller um = context.createUnmarshaller();
Schema schema = SchemaFactory.newInstance(W3C_XML_SCHEMA_NS_URI)
  .newSchema(Main.class.getResource("schema1.xsd"));
um.setSchema(schema);
Object bce = um.unmarshal(new File("src/conf/bcard.xml"));
m.marshal(bce, System.out);
```

Create an Unmarshaller through the generated schema file, and unmarshal the marshalled bcard.xml file and print it (marshal) to the console.

19.3. Java Standard API for XML Processing (JAXP)

Since XML and Java are platform independent, they are used together in various instances. JEUS 9 web services support Java standard API for handling XML documents. The standard is compliant with the standard (http://jcp.org/en/jsr/detail?id=206) for handling XML documents from JCP, a Java standard organization.

The following are the key JAXP APIs.

- SAX
- DOM
- TrAX
- DOM
- StAX

19.3.1. Java Streaming APIs for XML Parser (StAX)

JEUS 9 web services also support a Java streaming method that is used to parse XMLs. It is compliant with the API standard(http://www.jcp.org/en/jsr/detail?id=173) for Java streaming XML parser from JCP, a Java standard organization.