Eclipse Implementation of XML Web Services Release Documentation

Eclipse Implementation of XML Web Services Release Documentation

Table of Contents

Overview

Table of Contents

Eclipse Implementation of XML Web Services is a Web Services framework that provides tools and infrastructure to develop Web Services solutions for the end users and middleware developers.

With Eclipse Implementation of XML Web Services, clients and web services have a big advantage: the platform independence of the Java programming language.

1. Documentation

This release includes the following documentation:

- *Release Notes*
- *Users Guide*
- *Tools*
- API Documentation [../javadocs/rt]

The documentation/samples included with Eclipse Implementation of XML Web Services that is distributed on GitHub describes how to use Jakarta XML Web Services to develop and deploy your webservices endpoints in a servlet container. The samples discussed in this document and available as part of Eclipse Implementation of XML Web Services distribution are tested to work on GlassFish and on Apache Tomcat. The deployment is demonstrated using Eclipse Implementation of XML Web Services specific proprietary deployment descriptor sun-jaxws.xml.

If you wish to use Eclipse Implementation of XML Web Services in a Jakarta EE container in a portable manner (using Jakarta Enterprise Web Services deployment descriptors), please refer to the Jakarta EE 8 [https://jakarta.ee/specifications/platform/8/] or GlassFish [https://eclipse-ee4j.github.io/glassfish/] documentation/samples. The majority of the documentation included with Eclipse Implementation of XML Web Services is valid with Jakarta EE 8 as well.

2. Software Licenses

- 2012 Oracle Corporation and/or its affiliates. All rights reserved. [Copyright.txt]
- Eclipse Implementation of XML Web Services 2.3.3 is covered by EDL - version 1.0 [https://www.eclipse.org/org/documents/edl-v10.php]
- 3rd Party License README [ThirdPartyLicense.txt]

3. Useful Links

- Please use the Metro and Jakarta XML Binding [https://projects.eclipse.org/projects/ee4j.metro] forum for feedback.

- Eclipse Implementation of XML Web Services has a project page on GitHub: https://github.com/eclipse-ee4j/metro-jax-ws.
- Jakarta XML Web Services Group on Eclipse projects [https://projects.eclipse.org/projects/ee4j.metro]
- Issue Tracker [https://github.com/eclipse-ee4j/metro-jax-ws/issues]

Release Notes

Table of Contents

1. Required Software

- Requires Java SE 8 or later
- Requires Ant 1.9.7 [http://ant.apache.org/bindownload.cgi] or later
- Tested with GlassFish v5.x and Apache Tomcat 5.5.20

2. Installation Instructions

The following applies to the Eclipse Implementation of XML Web Services standalone bundles.

- Extract Eclipse Implementation of XML Web Services 2.3.3 bundle
- To use with GlassFish
  - GlassFish contains Eclipse Implementation of XML Web Services within Metro package so no installation required. For updating Eclipse Implementation of XML Web Services installation in GlassFish please refer to Metro [http://eclipselink-ee4j.github.io/metro-wsit] project.
- To install on Tomcat
  - Set `CATALINA_HOME` to your Tomcat installation.
  - Run

    ```
    ant install
    ```

    This essentially copies `lib/*.jar` from the root of Eclipse Implementation of XML Web Services extracted bundle to `$CATALINA_HOME/shared/lib`
  - If you are running on top of JDK8, run

    ```
    ant install-api
    ```

    or use

    ```
    ant -help
    ```

    for detailed information.

If you intend to only invoke an already existing Web service, then alternatively you can use the **wsimport** and **wsgen** scripts in the `bin` directory of the Eclipse Implementation of XML Web Services 2.3.3 bundle.

3. Running on top of JDK 8

JDK includes Jakarta XML Web Services and JAXB API and RI. Eclipse Implementation of XML Web Services 2.3.3 includes RI of Jakarta XML Web Services/JAXB 2.3. In order to run Eclipse Implementation of XML Web Services 2.3.3 you would need to specify ONLY `jakarta.xml.ws-api.jar`, and `jakarta.xml.bind-api.jar` jars using the Java Endorsed Standards Override Mechanism [https://docs.oracle.com/javase/8/docs/technotes/guides/standards/]

4. Jar dependency

Table 1. Jar dependencies summary

Runtime Jars	`FastInfoset.jar`, `jakarta.xml.bind-api.jar`, `jaxb-impl.jar`, `jakarta.xml.ws-api.jar`, `jakarta.annotation-api.jar`, `jaxws-rt.jar`, `jakarta.jws-api.jar`, `mimepull.jar`, `javax.xml.soap-api.jar`, `saaj-impl.jar`, `stax2-api.jar`, `woodstox-core.jar`, `stax-ex.jar`, `streambuffer.jar`, `policy.jar`, `gmbal.jar`, `management-api.jar`, `ha-api.jar`
Tooltime Jars	All the runtime jars + `jaxb-xjc.jar`, `jaxb-jxc.jar`, `jaxws-tools.jar`

If you run Eclipse Implementation of XML Web Services 2.3.3 with IBM JDK, there are two options:

- to add Oracles's JAXP implementation jars along
- to remove the saaj-impl.jar from the libraries in order IBM SAAJ implementation to be used

Eclipse Implementation of XML Web Services also depends on resolver implementation form JDK, which is not found on AIX. The repackaged resolver.jar can be found in the distribution, or Maven [http://search.maven.org/#search%7Cga%7C1%7Cg%3A%22com.sun.org.apache.xml.internal%22] though. First approach is recommended.

5. Current Limitations / Bugs / Issues

- The `java.util.Collection` classes cannot be used with rpc/literal or document/literal `BARE` style due to a limitation in JAXB. However, they do work in the default document/literal `WRAPPED` style.
- Although Jakarta XML Web Services customizations are portable across implementations, the names of WSDL and schema files generated are not specified. Therefore each vendor may and most likely will generate different names. Therefore, when switching between implementations of Jakarta XML Web Services, it may be necessary to modify your customization files to reflect different WSDL and schema file names.
- This holds true for different releases of the Eclipse Implementation of XML Web Services. The Eclipse Implementation of XML Web Services may in each release change these file names to resolve bugs. If and when this does occur, a developer would need to change the filenames in their customization files to match the new file names.

6. Changelog

- 2.2.8

```
** Bug
    * [JAX_WS-692] - WsImport fails if wsdl:message/wsdl:part defines
 "type" (not "element")
    * [JAX_WS-713] - Document MTOM limitations
    * [JAX_WS-1062] - wsimport command will throw NullPointerException when
 no difination of like "xmlns:undns="http://test"" in WSDL file.
    * [JAX_WS-1064] - Need to use Filer when generating files
    * [JAX_WS-1068] - Issue with two or more web services in the same war
 when pointing to wsdls in different META-INF/wsdl subdirs where the wsdl
 files themselves are the same.
    * [JAX_WS-1074] - ClassCast exception when wsimport task run in a
 forked mode
    * [JAX_WS-1083] -  Error listenerStart Sep 27, 2012 12:02:48 AM
 org.apache.catalina.core.StandardContext start
    * [JAX_WS-1087] - unable to delete .war file after wsimport completed
    * [JAX_WS-1092] - Back Compatible issue, for method:
 com.sun.xml.ws.server.EndpointFactory.verifyImplementorClass
    * [JAX_WS-1095] - Basic Authentication with wsimport does not allow @
 in username
    * [JAX_WS-1098] - IllegalAnnotationsException: 2 counts of
 IllegalAnnotationExceptions: MemberSubmissionEndpointReference$Address and
 W3CEndpointReference$Address
    * [JAX_WS-1099] -
 com.sun.xml.ws.api.model.wsdl.WSDLModel.WSDLParser.parse error in
 parsering  wsdl:message/wsdl:part defines "type" (not "element")
    * [JAX_WS-1101] - wsimport authFile URL containing passwords should
 support encoded/escaped characters...
    * [JAX_WS-1105] - wsgen fails to resolve all 'service implementation
 bean' methods
    * [JAX_WS-1107] - httpproxy  username and password not supported
    * [JAX_WS-1118] - Broken links in the guide

** Improvement
    * [JAX_WS-143] - wsimport does not support jaxb plugins
    * [JAX_WS-261] - Make WSServletDelegate class public
    * [JAX_WS-1100] - Allow wild card matching to allow the same
 user:password for all urls with the same host name
    * [JAX_WS-1102] - jaxws should pass encoding option to jaxb
    * [JAX_WS-1112] - Make JAX-WS run on Java SE 5.0 - fixed documentation
 to not refer to JDK5

** Task
    * [JAX_WS-1042] - The documentation in http://jax-ws.java.net/2.2.5/
docs/wsgenant.html is incorrect
    * [JAX_WS-1080] - move build from ant to maven
    * [JAX_WS-1082] - consider updating dependencies
```

  - See Section 18, “External Web Service Metadata” for more information on External metadata support.
  - JAXB version updated to 2.2.7, for changes see JAXB Changelog [https://javaee.github.io/jaxb-v2/doc/user-guide/ch02.html#a-2-2-7]

- 2.2.7 - 2.2.3 - TBD to be added
- 2.2.3
  - Add -clientjar option for **wsimport** [https://github.com/eclipse-ee4j/metro-jax-ws/issues/871]

- Add support for Async Servlet Transport using Servlet 3.0 api [https://github.com/eclipse-ee4j/metro-jax-ws/issues/872]
- Bug fixes [https://github.com/eclipse-ee4j/metro-jax-ws/issues?q=is%3Aissue+milestone%3A2.2.2+is%3Aclosed]

- 2.2.1
  - wsgen can inline schemas in a generated wsdl using a switch [https://github.com/eclipse-ee4j/metro-jax-ws/issues/85]
  - Schema Validation support is enhanced [https://github.com/eclipse-ee4j/metro-jax-ws/issues/777]
  - Bug fixes [https://github.com/eclipse-ee4j/metro-jax-ws/issues?q=is%3Aissue+is%3Aclosed+milestone%3A2.2]
- 2.2
  - Add WS-Addressing 1.0 Metadata support as per 2.2 spec
  - Support for `@XmlElement` on SEI's wrapper parameter
  - Support for `@XmlType` on exception classes
  - Implement HTTP SPI
  - Implement Endpoint API with features
  - Generate new constructors in the generated Service class(service level features, wsdllocation) [https://github.com/eclipse-ee4j/metro-jax-ws/issues/818]
  - Add support for customizing wrapper `wsdl:part` names as defined in 2.2 [https://github.com/eclipse-ee4j/metro-jax-ws/issues/726]
  - Add support for customizing `wsdl:message` name using `@WebFault` as defined in 2.2 [https://github.com/eclipse-ee4j/metro-jax-ws/issues/727]
  - Fault customization should accept fully qualified name of the exception class to be generated [https://github.com/eclipse-ee4j/metro-jax-ws/issues/728]
  - Customization of Service should accept fully qualified class name [https://github.com/eclipse-ee4j/metro-jax-ws/issues/729]
  - Add Xnocompile option for **wsgen** similar to **wsimport** [https://github.com/eclipse-ee4j/metro-jax-ws/issues/360]
  - XPath expressions in jax-ws customization can now select multiple nodes [https://github.com/eclipse-ee4j/metro-jax-ws/issues/576]
  - Disable server-side exception stacktrace for service specific exceptions [https://github.com/eclipse-ee4j/metro-jax-ws/issues/761]
  - Optimzation of LogicalMessage implementation [https://github.com/eclipse-ee4j/metro-jax-ws/issues/648]
  - wsimport now handles URL redirection of imported wsdls correctly [https://github.com/eclipse-ee4j/metro-jax-ws/issues/305]

    - Bug fixes [https://github.com/eclipse-ee4j/metro-jax-ws/issues?q=is%3Aissue+is%3Aclosed+milestone%3A2.1.7]

- 2.1.7

    - Bug fixes [https://github.com/eclipse-ee4j/metro-jax-ws/issues?q=is%3Aissue+is%3Aclosed+milestone%3A2.1.6]

- 2.1.6

    - Supportfor transport attribute for bindings like SOAP/JMS binding [https://github.com/eclipse-ee4j/metro-jax-ws/issues/741]

    - Bug fixes [https://github.com/eclipse-ee4j/metro-jax-ws/issues?q=is%3Aissue+is%3Aclosed+milestone%3A2.1.5]

- 2.1.5

    - Allowing HTTP response code to be set via MessageContext property [https://github.com/eclipse-ee4j/metro-jax-ws/issues/607]

    - New feature: UsesJAXBContextFeature, to control JAXBContext creation [https://github.com/eclipse-ee4j/metro-jax-ws/issues/2822]

    - New **wsimport** option: -XdisableSSLHostnameVerfication, to disable SSL Hostname verification [https://github.com/eclipse-ee4j/metro-jax-ws/issues/289]

    - Wsimport ant task accepts nested args for specifying additional options [https://github.com/eclipse-ee4j/metro-jax-ws/issues/490]

    - Bug fixes [https://github.com/eclipse-ee4j/metro-jax-ws/issues?q=is%3Aissue+is%3Aclosed+milestone%3A2.1.4]

- 2.1.4

    - Simplified "Starting from Java Programming model" through dynamic generation of request and response wrappers

    - Support for dual binding (SOAP/HTTP and XML/HTTP) for the same service class

    - Bug fixes [https://github.com/eclipse-ee4j/metro-jax-ws/issues?q=is%3Aissue+is%3Aclosed+milestone%3A2.1.3]

- 2.1.3

    - Authentication support in **wsimport** [https://github.com/eclipse-ee4j/metro-jax-ws/issues/231]

    - Additional header support [https://github.com/eclipse-ee4j/metro-jax-ws/issues/263]

    - Large and Streaming attachment support [https://github.com/eclipse-ee4j/metro-jax-ws/issues/29]

    - XML Schema validation support for SOAP message [https://github.com/eclipse-ee4j/metro-jax-ws/issues/239]

    - Expose the -Xnocompile as an Ant task option [473]

    - Additional WSDL object methods for policy [https://github.com/eclipse-ee4j/metro-jax-ws/issues/437]
    - Wsimport should be able to handle redirects and see Others [https://github.com/eclipse-ee4j/metro-jax-ws/issues/456]
    - Remove SJSXP and SAAJ RI dependency [https://github.com/eclipse-ee4j/metro-jax-ws/issues/54]
    - Wsimport should write to passed OutputStream [https://github.com/eclipse-ee4j/metro-jax-ws/issues/387]
    - Efficient Handler using Handler<Message> support [https://github.com/eclipse-ee4j/metro-jax-ws/issues/482]
    - Bug fixes [https://github.com/eclipse-ee4j/metro-jax-ws/issues?q=is%3Aissue+is%3Aclosed+milestone%3A2.1.2]
- 2.1.2
    - JMX Agent for the server side
    - Mtom Interop with .NET 2.0/WSE 3.0
    - Bug fixes [https://github.com/eclipse-ee4j/metro-jax-ws/issues?q=is%3Aissue+is%3Aclosed+milestone%3A2.1.1]
- 2.1.1
    - JAXB 2.1 RI Integration
    - JAX-WS 2.1 MR (JSR 224) implementation
    - Type substitution support
    - WS-Addressing - W3C and Memeber Submission
    - APIs to create EPR and use it to invoke endpoint
    - Support for enabling/disabling features, such as WS-Addressing/MTOM
    - Asynchronous server
    - Tubes and Fiber for asynchronous message processing
    - Dispatch<Message> and Provider<Message>
    - Stateful Webservice support
    - Bug fixes [https://github.com/eclipse-ee4j/metro-jax-ws/issues?q=is%3Aissue+is%3Aclosed+milestone%3A2.1]

Users Guide

Table of Contents

1. Overview

1.1. Introduction

This document describes the new features available in this release of the Eclipse Implementation of XML Web Services. The main focus of this document is to describe the tools used to develop Eclipse Implementation of XML Web Services 2.3.3 web service endpoints and clients. Readers of this document should be familiar with web services XML [http://www.w3.org/TR/2000/REC-xml-20001006], XML Schema [http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/] and WSDL [http://www.w3.org/TR/2001/NOTE-wsdl-20010315]. Familiarity with Jakarta XML RPC 1.1 [https://jakarta.ee/specifications/xml-rpc/] may also be beneficial but is not necessary.

The documentation/samples discusses how to use Eclipse Implementation of XML Web Services in a non-Jakarta EE servlet container using a proprietary deployment descriptor `sun-jaxws.xml` and servlet `com.sun.xml.ws.transport.http.servlet.WSServlet`. This means that you can run Eclipse Implementation of XML Web Services applications in any servlet container that has been enabled with the Eclipse Implementation of XML Web Services. Applications that use the proprietary DD and servlet will run in a Eclipse Implementation of XML Web Services enabled Jakarta EE servlet container, but they will be non-portable. If you wish to use Jakarta XML Web Services in a Jakarta EE container in

a Jakarta EE portable manner you need to use the standard Jakarta EE deployment descriptor; please refer to the Jakarta EE [https://jakarta.ee/specifications/platform/] or GlassFish [https://eclipse-ee4j.github.io/glassfish/] documentation/samples. The majority of the documentation included with Jakarta XML Web Services is valid with Jakarta EE as well.

1.2. Features

1.2.1. Jakarta XML Web Services $ws.spec.version; API

Jakarta XML Web Services $ws.spec.version; is a Maintainence Release of JAXWS 2.0 API.

Jakarta XML Web Services $ws.spec.version; has the following new features from Jakarta XML Web Services 2.1 specification:

- Support for Jakarta XML Binding APIs
- Support for WS-Addressing 1.0 - Metadata specification
- Support for `@XmlElement` on SEI's wrapper parameter
- Support for `@XmlType` on exception classes
- HTTP SPI
- Provide API to create Endpoint with features

Jakarta XML Web Services 2.1 has the following new features from Jakarta XML Web Services 2.0 specification:

- WS-Addressing support
- APIs for EndpointReference
  - Creation
    - `BindingProvider.getEndpointReference()`
    - `Endpoint.getEndpointReference()`
    - `MessageContext.getEndpointReference()`
  - EPR Propagation
    - Using Jakarta XML Binding 2.1 bind W3C EPR to `W3CEndpointReference` class
    - Marshall/Unmarshall `W3CEndpointReference` class using Jakarta XML Binding
- User friendly APIs to enable/disable features, such as MTOM and Addressing

JAX-RPC users should note that Jakarta XML Web Services is a completely different technology than JAX-RPC and thus cannot run JAX-RPC applications on top of Jakarta XML Web Services. If you have an existing JAX-RPC application it must be converted to work with Jakarta XML Web Services.

1.2.2. Fully Dynamic Runtime

In Jakarta XML Web Services, all artifacts generated by **annotationProcessing**, **wsimport** and **wsgen** are portable. Jakarta XML Web Services uses the annotations within the SEI to aid in marshalling/unmar-

shalling messages. Because we no longer generated non-portable artifacts, we were able to get rid of tools like JAX-RPC's **wsdeploy**. The user now can create their own deployable WAR file. To learn more about creating a WAR file and the deployment descriptor, see WAR File Packaging. It should also be noted that JAX-RPC's **wscompile** tool has been replaced by two new tools: **wsimport** and **wsgen**. **wsimport** is used for importing WSDLs and generating the portable artifacts. **wsgen** processes a compiled SEI and generates the portable artifacts. Unlike JAX-RPC's **wscompile** Jakarta XML Web Services's **wsgen** does not generate WSDL at tool-time, the WSDL is now generated when the endpoint is deployed. There however is an option on **wsgen** to generate the WSDL for developement purposes.

1.2.3. MTOM & swaRef

MTOM and swaRef support was added in Eclipse Implementation of XML Web Services release. MTOM and swaref support is required by the Jakarta XML Web Services specification. This means that the MTOM or swaref solution developed with Eclipse Implementation of XML Web Services will be fully portable with any Jakarta XML Web Services compliant implementation.

MTOM implementation was completely re-written to allow streaming attachment support and just like rest of the Eclipse Implementation of XML Web Services runtime its written for better performance. This implementation was released as part of Eclipse Implementation of XML Web Services release.

Eclipse Implementation of XML Web Services brings in support for optimized transmission of binary data as specified by MTOM [http://www.w3.org/TR/soap12-mtom/] (SOAP Message Transmission Optimization Mechanism)/ XOP [http://www.w3.org/TR/xop10/] (XML Binary Optimized Packaing) and swaRef [http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html#Example_Attachment_Description_Using_swaRef] (SOAP Attachment References specified by WS-I Attachment Profile 1.0).

- MTOM allows optimized transmission of binary data - any `xs:base64Binary` or `xs:hexBinary` schema type can be send as attachment following rules defined by XOP encoding and MTOM specification.

- In swaRef, an XML element of `wsi:swaRef` [http://ws-i.org/profiles/basic/1.1/xsd/] type (defined by WS-I Attachment Profile 1.0) is send as attachment and a referenced based on CID URL schema is put in place of the content of the element.

For details on MTOM and swaRef features refer to MTOM and Swaref.

1.2.4. SOAP 1.2

SOAP 1.2 support is added to Eclipse Implementation of XML Web Services. For details refer to SOAP 1.2.

1.2.5. XML/HTTP Binding

Support for XML/HTTP binding is added to Jakarta XML Web Services. One can directly send XML over HTTP using Provider and Dispatch implementations. This enables support for REST style Web Services in Jakarta XML Web Services. For details refer to restful sample.

1.2.6. Jakarta XML Binding 2.3

Eclipse Implementation of XML Web Services 2.3.3 uses Jakarta XML Binding 2.3 for data-binding between Java and XML which enables features such as separate compilation, type substitution and other improvements.

1.2.6.1. Type Substitution using `@XmlSeeAlso`

Jakarta XML Binding 2.1 defines `@XmlSeeAlso` annotation which can be used to tell Jakarta XML Binding to use the classes mentioned with this annotation. This allows type substitution to takes place. See the `samples/type_substitution/src/type_substitution/server/CarDealer.java` sample that demonstrates it.

wsimport tool, generates `@XmlSeeAlso` with all the classes that are not directly referenced by the WSDL operations. To capture all such classes **wsimport** generates `@XmlSeeAlso(ObjectFactory.class)` on the generated Service Endpoint Interface.

1.2.6.2. `@XmlElement` on web service SEI parameters

Jakarta XML Web Services 2.2 spec allows `@XmlElement` on web service SEI parameters, which enables better control of XML representation. For this support, Jakarta XML Web Services relies on Jakarta XML Binding 2.2 API which allows the `@XmlElement` annotation on parameters.

1.2.7. WS-Addressing

Eclipse Implementation of XML Web Services 2.3.3 supports for W3C Core [http://www.w3.org/TR/ws-addr-core], SOAP Binding [http://www.w3.org/TR/ws-addr-soap] and Addressing 1.0 - Metadata [http://www.w3.org/TR/2007/REC-ws-addr-metadata-20070904] specifications and defines standard API and annotations to enable/disable W3C WS-Addressing on the client and service endpoint. In addition to that, Eclipse Implementation of XML Web Services also supports Member Submission [http://www.w3.org/Submission/ws-addressing/] version of WS-Addressing. The member submission version is supported in an implementation specific way. For compatility with Jakarta XML Web Services 2.1 behavior, Eclipse Implementation of XML Web Services 2.2 also supports wsdls conforming to WSDL Binding [http://www.w3.org/TR/ws-addr-wsdl] specification.

Refer to WS-Addressing for more details. See WS-Addressing samples **fromjava-wsaddressing**, **fromwsdl-wsaddressing-policy** and **fromwsdl-wsaddressing** with the Eclipse Implementation of XML Web Services 2.3.3 for details on the WS-Addressing programming model.

1.2.8. Annotations

Jakarta XML Web Services $ws.spec.version; relies heavily on the use of annotations as provided by *A Metadata Facility for the Java Programming Language* (JSR 175) [http://jcp.org/en/jsr/detail?id=175] and *Jakarta Web Services Metadata* (link) [https://jakarta.ee/specifications/web-services-metadata/] as well as additional annotations defined by Jakarta XML Web Services $ws.spec.version;. These annotations are used to customize the mapping from Java to XML schema/WSDL and are used at runtime to alleviate the need for non-portable serializers/deserializers that were generated in JAX-RPC 1.x. (JSR 269) [http://jcp.org/en/jsr/detail?id=269] Pluggable Annotation Processing API comes as replacement of apt

The Eclipse Implementation of XML Web Services utilizes an **javac** Pluggable Annotation Processing API functionality that was introduced in Java SE 6. **javac** allows the SI to process Java source files directly to generate the portable artifacts specified by the Jakarta XML Web Services specification. **javac** comes as replacement of deprecated **apt**. More documentation about **javac** can be found in section **Annotation Processing** Deprecated **apt** will be covered in more detail in section **apt**.

For more information on the annotations used by Jakarta XML Web Services 2.0 please refer to Annotations.

1.2.9. Customizations

Eclipse Implementation of XML Web Services 2.3.3 carries forward customization support introduced in Jakarta XML Web Services.

- Define a package where Java artifacts mapped from a WSDL file will be generated
- Package customization for value classes mapped from the imported XML schemas by the WSDL document
- Handler chain customization

Jakarta XML Web Services specification defines standard XML based customization for a WSDL file to Java mapping and to control certain features. These customizations, or *binding declarations*, can customize almost all WSDL components that can be mapped to Java, such as the service endpoint interface class, method name, parameter name, exception class, etc. The other important thing you can do with these binding declarations is to control certain features, such as asynchrony, provider, wrapper style, and additional headers. For example, a client application can enable asynchrony for a particular operation in a portType, all operations in a `portType`, or all `portType` operations defined in the WSDL file.

These binding declarations can be inlined in a WSDL file or can live outside as an external file. The binding declarations closely align with the Jakarta XML Binding binding declarations. An application importing a WSDL file can inline Jakarta XML Binding bindings inside Jakarta XML Web Services binding declarations to customize the inlined schema declared in the WSDL file. Schema files that are imported from a WSDL file can be customized using Jakarta XML Binding binding files and can be passed to **wscompile** using the `-b` option switch.

These are the main customization features:

- Scoped binding declarations. An XPath expression is used to specify the target node of the WSDL file on which customization should be applied.
- Close alignment with Jakarta XML Binding bindings. Jakarta XML Binding binding declarations can be inlined in an external Jakarta XML Web Services binding file.
- Feature Control. Features such as asynchrony, wrapper style, additional header mapping, and provider interfaces can be enabled or disabled.
- Handler chain customization (not yet specified by the 2.0 specification)

The following WSDL component's mapped Java names can be modified:

- generated service endpoint interface class
- method
- method parameter
- generated exception class (for WSDL fault and header fault exceptions)
- header parameter
- generated service class
- port accessor methods in the generated service class

XML Schema Java mapping can be customized using standard Jakarta XML Binding customizations.

For more information on the customizations used by Jakarta XML Web Services please refer to WSDL Customization.

1.2.10. Improved Handler Framework

Jakarta XML Web Services specification defines two types of handlers: logical and protocol handlers. While protocol handlers have access to an entire message such as a SOAP message, logical handlers deal only with the payload of a message and are independent of the protocol being used. Handler chains can now be configured on a per-port, per-protocol, or per-service basis. A new framework of context objects has been added to allow client code to share information easily with handlers.

For more information on the handler framework in Eclipse Implementation of XML Web Services 2.3.3 please refer to Handler.

1.2.11. Provider

Web service endpoints may choose to work at the XML message level by implementing the `Provider` interface. Here the endpoints access messages or message payloads using this low level, generic API.

For more information on providers in Eclipse Implementation of XML Web Services 2.3.3 please refer to Provider.

1.2.12. Dispatch

The Dispatch API is intended for advanced XML developers who prefer to use XML constructs at the `java.lang.transform.Source` or `javax.xml.soap.SOAPMessage` level. For added convenience use of the `Dispatch` API with Jakarta XML Binding data-bound objects is supported. The `Dispatch` API can be used in both `Message` and `Payload` modes.

For more information on the `Dispatch` please refer to Dispatch.

1.2.13. Asynchronous Client Operations

For more information on asynchronous clients in Eclipse Implementation of XML Web Services 2.3.3 please refer to Asynchronous Client.

1.3. Programming Model

This section of the documentation will focus on the programming model for both developing and publishing a web service endpoint, and writing a web service client. A web service endpoint is the implementation of a web service. A web service client is an application that accesses a web service.

1.3.1. Server

When developing a web service endpoint, a developer may either start from a Java endpoint implementation class or from a WSDL file. A WSDL (Web Services Description Language) document describes the contract between the web service endpoint and the client. A WSDL document may include and/or import XML schema files used to describe the data types used by the web service. When starting from a Java class, the tools generate any portable artifacts as mandated by the spec. When starting from a WSDL file and schemas, the tools generate a service endpoint interface.

There is a trade-off when starting from a Java class or from a WSDL file. If you start from a Java class, you can make sure that the endpoint implementation class has the desirable Java data types, but the developer has less control of the generated XML schema. When starting from a WSDL file and schema, the developer has total control over what XML schema is used, but has less control over what the generated service endpoint and the classes it uses will contain.

1.3.1.1. Starting from Java

The basic process for deploying a web service from a Java class consists of two steps.

1. Generate portable artifacts.

2. Create a WAR file to deploy

1.3.1.2. Generate Portable Artifacts

Portable artifacts generated by Eclipse Implementation of XML Web Services 2.3.3 include zero or more JavaBean classes to aide in the marshaling of method invocations and responses, as well as service-specific exceptions.

In document/literal wrapped mode, two JavaBeans are generated for each operation in the web service. One bean is for invoking the other for the response. In all modes (rpc/literal and both document/literal modes), one JavaBean is generated for each service-specific exception.

When starting from Java the developer must provide the Jakarta XML Web Services tools with a valid endpoint implementation class. This implementation class is the class that implements the desired web service. Jakarta XML Web Services has a number of restrictions on endpoint implementation classes. A valid endpoint implementation class must meet the following requirements:

- It *must* carry a javax.jws.WebService annotation (see Jakarta Web Services Metadata).
- Any of its methods *may* carry a javax.jws.WebMethod annotation (see 7.5.2).
- All of its methods *may* throw java.rmi.RemoteException in addition to any service-specific exceptions.
- All method parameters and return types *must* be compatible with the Jakarta XML Binding 2.0 Java to XML Schema mapping definition.
- A method parameter or return value type *must not* implement the java.rmi.Remote interface either directly or indirectly.

Here is an example of a a simple endpoint implementation class `samples/fromjava/src/fromjava/server/AddNumbersImpl.java` from the fromjava sample:

```
package fromjava.server;

import javax.jws.WebService;

@WebService
public class AddNumbersImpl {
    /**
     * @param number1
     * @param number2
     * @return The sum
     * @throws AddNumbersException if any of the numbers to be added is
     * negative.
     */
    public int addNumbers(int number1, int number2) throws
            AddNumbersException {
        if (number1 < 0 || number2 < 0) {
            throw new AddNumbersException("Negative number cant be " +
                    "added!", "Numbers: " + number1 + ", " + number2);
        }
        return number1 + number2;
```

```
    }
}
```

If you are familiar with JAX-RPC 1.1, you will notice that this implementation class does not implement a service endpoint interface. In Eclipse Implementation of XML Web Services 2.3.3 a service endpoint interface is no longer required.

When starting from a Java endpoint implementation class, it is recommended that the portable artifacts be generated from source using **annotationProcessing**. This because the Jakarta XML Web Services tools will then have full access to the source code and will be able to utilize parameter names that are otherwise not available through the Java reflection APIs. If the source for the endpoint implementation class is not available, the portable artifacts can be generated using **wscompile**. Here is a sample **annotationProcessing** Ant task from the samples:

```
<annotationProcessing
        debug="${debug}"
        verbose="${verbose}"
        destdir="${build.classes.home}"
        sourceDestDir="${build.classes.home}"
        srcdir="${basedir}/src"
        includes="**/server/*.java"
        sourcepath="${basedir}/src">

    <classpath refid="jax-ws.classpath"/>
</annotationProcessing>
```

More information about the **annotationProcessing** Ant task can be found **annotationProcessing** Ant Task. If this task is run on the fromjava sample, the output would include:

```
AddNumbers.class
AddNumbers.java
AddNumbersExceptionBean.class
AddNumbersExceptionBean.java
AddNumbersResponse.class
AddNumbersResponse.java
```

The `AddNumbersImplService.wsdl` file describes the web service. The `schema1.xsd` file is imported by the `AddNumbersImplService.wsdl` and contains the datatypes used by the web service. The `AddNumbers.class`/`AddNumbers.java` files contain the a bean used by a Jakarta XML Binding to marshall/unmarshall the `addNumbers` request. The `AddNumbersExceptionBean.class`/`AddNumbersExceptionBean.java` file is a bean used by Jakarta XML Binding to marshall the contents of the `AddNumbersException` class. The `AddNumbersResponse.class`/`AddNumbersResponse.java` files represent the response bean used by Jakarta XML Binding to marshall/unmarshall the `addNumbers` response.

1.3.1.3. Create a WAR file to deploy

Creating a WAR file is nothing more than packaging the service endpoint interface (if there is one), service endpoint implementation, Java classes used by the endpoint implementation and a deployment descriptor in WAR format. For the fromjava sample the `AddNumbersImpl` and `AddNumbersException` classes in the fromjava.server package, and the deployment descriptor are bundled together to make a raw WAR file. To learn more about creating a WAR file and the deployment descriptor, see WAR File Packaging. The deployment descriptor used in fromjava sample is given below and can be found `samples/fromjava/etc/sun-jaxws.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
        version='2.0'>
```

```
    <endpoint name='fromjava'
            implementation='fromjava.server.AddNumbersImpl'
            url-pattern='/addnumbers'/>
</endpoints>
```

The attributes of the `<endpoint>` element are described below:

- name is simply an identifier for this endpoint
- implementation is used to specify the endpoint implementation class
- urlpattern is used to URL pattern used to access this endpoint.

The structure of the raw WAR file is shown below:

```
META-INF/MANIFEST.MF
WEB-INF/sun-jaxws.xml
WEB-INF/web.xml
WEB-INF/classes/fromjava/server/AddNumbersException.class
WEB-INF/classes/fromjava/server/AddNumbersImpl.class
WEB-INF/classes/fromjava/server/jaxws/AddNumbers.class
WEB-INF/classes/fromjava/server/jaxws/AddNumbersExceptionBean.class
WEB-INF/classes/fromjava/server/jaxws/AddNumbersResponse.class
```

The WAR file created can now be published on a Eclipse Implementation of XML Web Services enabled servlet container such as the Sun Java System Application Server Platform Edition 8.2 [http://java.sun.com/j2ee/1.4/download.html]

1.3.1.4. Starting from a WSDL File

The basic process for deploying a web service when starting from a WSDL document consists of the following four steps:

1. Generate a service endpoint interface.
2. Implement the service endpoint interface.
3. Create a WAR file to deploy.

1.3.1.5. Generate a Service Endpoint Interface

This step involves compiling or importing the WSDL file to generate a service endpoint interface and value classes mapped from imported XML schemas.

Below is a sample **wsimport** Ant target:

```
<wsimport
        debug="${debug}"
        verbose="${verbose}"
        keep="${keep}"
        destdir="${build.classes.home}"
        wsdl="${server.wsdl}">

    <binding dir="${basedir}/etc"
            includes="${server.binding}"/>
</wsimport>
```

Its commandline equivalent is:

```
wsimport.sh etc/AddNumbers.wsdl -b custom-server.xml
```

Lets look at the excerpt of `samples/fromwsdl/etc/AddNumbers.wsdl` from the sample fromwsdl:

The generated service endpoint interface looks as follows:

```
package fromwsdl.server;

@javax.jws.WebService(
        name = "AddNumbersPortType",
        serviceName = "AddNumbersService",
        targetNamespace = "http://duke.example.org")
@javax.jws.soap.SOAPBinding(
        style = javax.jws.soap.SOAPBinding.Style.DOCUMENT,
        use = javax.jws.soap.SOAPBinding.Use.LITERAL,
        parameterStyle = javax.jws.soap.SOAPBinding.ParameterStyle.WRAPPED)
public interface AddNumbersPortType extends java.rmi.Remote {
    @javax.jws.WebMethod(operationName = "addNumbers")
    @javax.jws.WebResult(name = "return")
    public int addNumbers(@javax.jws.WebParam(name = "arg0") int arg0,
                          @javax.jws.WebParam(name = "arg1") int arg1)
            throws fromwsdl.server.AddNumbersFault_Exception,
 java.rmi.RemoteException;
}
```

The generated service endpoint interface has annotations that can be used by the future versions of Jakarta XML Web Services to do dynamic binding and serialization/deserialization at runtime. Alternatively this service endpoint interface can be used to generate a WSDL and schema file. Please note that round-tripping is not guaranteed in this case. So the generated WSDL file and schema may not be the same as the one the service endpoint interface was generated from.

1.3.1.6. Implement the Service Endpoint Interface

The next thing to do will be to provide the implementation of the service endpoint interface generated in the previous step. When you implement the service endpoint interface it is necessary to provide a `@WebService` annotation on the implementation class with a endpointInteface element specifying the qualified name of the endpoint interface class. Let's look at the implementation class `samples/fromwsdl/src/fromwsdl/server/AddNumbersImpl.java` from the sample application fromwsdl:

```
package fromwsdl.server;

@WebService(endpointInterface = "fromwsdl.server.AddNumbersPortType")
public class AddNumbersImpl implements AddNumbersPortType {
    /**
     * @param number1
     * @param number2
     * @return The sum
     * @throws AddNumbersException if any of the numbers to be added is
     * negative.
     */
    public int addNumbers(int number1, int number2) throws
            AddNumbersFault_Exception {
        ...
    }
}
```

1.3.1.7. Create a WAR

This step is similar to the one described above in Create a WAR file to deploy .

Here the service endpoint interface implementation class from previous step, together with a deployment descriptor file sun-jaxws.xml, and `web.xml` should be bundled together with the service endpoint interface, value classes generated in the first step mentioned in Generate a Service Endpoint Interface.

Let's look at `samples/fromwsdl/etc/sun-jaxws.xml` from the sample application fromwsdl:

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
        version="2.0">
    <endpoint name="fromwsdl"
            interface="fromwsdl.server.AddNumbersPortType"
            implementation="fromwsdl.server.AddNumbersImpl"
            wsdl="WEB-INF/wsdl/AddNumbers.wsdl"
            service="{http://duke.example.org}AddNumbersService"
            port="{http://duke.example.org}AddNumbersPort"
            url-pattern="/addnumbers"/>
</endpoints>
```

It defines the deployment-related configuration information for the fromwsdl endpoint. You will notice that this deployment descriptor contains additional attributes than the deployment descriptor described in Create a WAR file to deploy. The interface attribute references the service endpoint interface generated in step 1. The wsdl attribute also points at the WSDL that was imported by **wsimport**. The service attribute references which service in the WSDL this endpoint is from and the port is the name of the port in that service for this endpoint.

To learn more about creating a WAR file and the deployment descriptor, see WAR File Packaging.

The WAR file created can now be published on a Eclipse Implementation of XML Web Services enabled servlet container such as the Sun Java System Application Server Platform Edition 8.2 [http://java.sun.com/j2ee/1.4/download.html]

1.3.1.8. Java SE Endpoints

Endpoints can be created and published programmatically using `javax.xml.ws.Endpoint` API in Java SE. To learn more about these endpoints, see Endpoint API.

1.3.2. Client

A client application can access a remote web service endpoint in one of two ways: port and dispatch.

1.3.2.1. Dynamic Proxy

In this approach client side invokes Web services via a dynamic proxy. The proxies for the Web Service are created from the generated Service and service endpoint interfaces. Once the proxies are created. the client application can invoke methods on those proxies just like a standard implementation of those interfaces. The sections below describe this process more detail.

1.3.2.2. Generate Client Artifacts

The **wsimport** tool is used to generate the service endpoint interface and the service interface classes. Below is the sample **wsimport** Ant target:

```
<wsimport
        debug="${debug}"
        verbose="${verbose}"
        keep="${keep}"
        destdir="${build.classes.home}"
        wsdl="${client.wsdl}">
```

```
    <classpath>
        <path refid="jax-ws.classpath"/>
        <pathelement location="${build.classes.home}"/>
    </classpath>
    <binding dir="${basedir}/etc" includes="${client.binding}"/>
</wsimport>
```

The command line equivalent of this Ant target is:

```
wsimport.sh -classpath client_classpath -d dest_dir -s src_dir \
    -b custom-client.xml http://localhost:8080/jax-ws-fromwsdl/addnumbers?
WSDL
```

For more details see the **wsimport** documentation.

Here is the excerpt from `samples/fromwsdl/src/fromwsdl/client/AddNumbersClient.java` in the fromjava sample application:

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

//invoke the remote method
int result = port.addNumbers(10, 20);
```

1.3.2.3. Dispatch

The `Dispatch` API is intended for advanced XML developers who prefer using XML constructs at the `java.lang.transform.Source` or `javax.xml.soap.SOAPMessage` level. For added convenience use of `Dispatch` with Jakarta XML Binding data binding object is supported. With the `XML/HTTP` binding a `javax.activation.DataSource` can also be used. The `Dispatch` APIs can be used in both `Message` and `Payload` modes. The `Dispatch` API client with an `XML/HTTP` binding can be used with REST Web Services. Please see the restful sample program for more information.

For more information on `Dispatch` in Eclipse Implementation of XML Web Services 2.3.3 please refer to Dispatch.

1.4. Useful Information

Pluggable Annotation Processing API [http://jcp.org/aboutJava/communityprocess/final/jsr269/] – http://jcp.org/aboutJava/communityprocess/final/jsr269/

Annotation Processing Tool (apt) [http://docs.oracle.com/javase/6/docs/technotes/guides/apt/index.html] – http://docs.oracle.com/javase/6/docs/technotes/guides/apt/index.html.

Please use the METRO [https://projects.eclipse.org/projects/ee4j.metro] forum for feedback.

The Eclipse Implementation of XML Web Services project on GitHub is: https://github.com/eclipse-ee4j/metro-jax-ws.

2. Provider

Web Service endpoints may choose to work at the XML message level by implementing the `Provider` interface. This is achieved by implementing either `Provider<Source>` or `Provider<SOAPMessage>` or `Provider<DataSource>`. The endpoint accesses the message or message payload using this low-level, generic API. All the Provider endpoints must have `@WebServiceProvider` annotation. The `@ServiceMode` annotation is used to convey whether the endpoint wants to access the message (`Service.Mode.MESSAGE`) or payload (`Service.Mode.PAYLOAD`).

If there is no `@ServiceMode` annotation on the endpoint, payload is the default value. The endpoint communicates with handlers using `WebServiceContext resource like any other normal endpoint`. Provider endpoints can start from java or WSDL. When the provider endpoints start from a WSDL file, `<provider>` WSDL customization can be used to mark a port as a provider.

2.1. `Provider<Source>` and `PAYLOAD`

An endpoint can access only the payload of a request using `Service.Mode.PAYLOAD` in the `@ServiceMode` annotation. This is the default behaviour, if the annotation is missing.

For example:

```
@WebServiceProvider
public class ProviderImpl implements Provider<Source> {
    public Source invoke(Source source) { // do request processing Source
        response = ...;
        return response;
    }
}
```

2.2. `Provider<SOAPMessage>` and `MESSAGE`

An endpoint can access an entire SOAP request as a `SOAPMessage`. `Service.Mode.MESSAGE` in the `@ServiceMode` annotation is used to convey the intent.

For example:

```
@WebServiceProvider
@ServiceMode(value = Service.Mode.MESSAGE)
public class ProviderImpl implements Provider<SOAPMessage> {
    public SOAPMessage invoke(SOAPMessage msg) { // do request processing
        SOAPMessage response =...;
        return response;
    }
}
```

2.3. `Provider<Source>` and `MESSAGE`

An endpoint can access a request as a `Source`. If the request is a `SOAPMessage`, only the `SOAPPart` (no attachments) of the message is passed as `Source` to the `invoke` method. If the returned response is null, it is considered a one way MEP.

For example:

```
@ServiceMode(value = Service.Mode.MESSAGE)
public class ProviderImpl implements Provider<Source> {
    public Source invoke(Source source) { // do request processing
        using source // return null to indicate oneway return null;
    }
}
```

2.4. WSDL Customization

If the provider endpoint starts with a WSDL file, a port can be customized to a provider endpoint using the `<provider>` customization. **wsimport** won't generate any artifacts for that port.

For example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bindings
        ...
        wsdlLocaption="AddNumbers.wsdl"xmlns="http://java.sun.com/xml/ns/
jaxws">
    <bindings node="wsdl:definitions">
        <package name="provider.server"/>
        <provider>true</provider>
    </bindings>
</bindings>
```

2.5. The `sun-jaxws.xml` file

The attributes of provider endpoint in sun-jaxws.xml: name, implementation, wsdl, service, port override the attributes provided through `@WebServiceProvider` annotation. For SOAP1.2 binding, one needs to specify binding attribute.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
        version='2.0'>
    <endpoint name='AddNumbers'
            implementation='provider.server.AddNumbersImpl'
            wsdl='WEB-INF/wsdl/AddNumbers.wsdl'
            service='{http://duke.example.org}AddNumbersService'
            port='{http://duke.example.org}AddNumbersPort'
            url-pattern='/addnumbers'/>
</endpoints>
```

If the wsdl, service, port are not specified in sun-jaxws.xml, then should be declared in the `@WebServiceProvider` annotation in implementation class.

2.6. Provider and Binding

Provider endpoint can be configured for different bindings using binding ids. These binding ids are defined in Jakarta XML Web Services API and endpoint can be configured by specifying `@BindingType` annotation or using binding attribute in sun-jaxws.xml. sun-jaxws.xml overwrites binding defined by `@BindingType` annotation. If the binding is not specified using `@BindingType` or in `sun-jaxws.xml`, the default binding is SOAP1.1/HTTP.

For example: To specify `XML/HTTP` binding using `@BindingType` annotation

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

For example: To specify `XML/HTTP` binding in `sun-jaxws.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
        version='2.0'>
    <endpoint
            ...
            binding="http://www.w3.org/2004/08/wsdl/http"/>
```

```
</endpoints>
```

2.7. RESTful Provider endpoints

RESTful Web Services can be built using `XML/HTTP` binding based `Provider` endpoints. In this case, even HTTP GET requests are passed to the endpoint. `Endpoint` can get necessary HTTP request query string and path information using standard `MessageContext.QUERY_STRING` and `MessageContext.PATH_INFO`. For more details on endpoint implementation, see the `samples/restful/src/restful/server/AddNumbersImpl.java` sample. If the endpoint expects GET requests to contain extra path after the endpoint address, then `url-pattern` should have "`/*`" at the end in both `sun-jaxws.xml` and `web.xml`.

For example: `sun-jaxws.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
        version='2.0'>
    <endpoint
            ...
            binding="http://www.w3.org/2004/08/wsdl/http"
            url-pattern="/addnumbers/*"/>
</endpoints>
```

For example: `web.xml`

```
<web-app>

    ...

    <servlet-mapping>
        <servlet-name>provider</servlet-name>
        <url-pattern>/addnumbers/*</url-pattern>
    </servlet-mapping>

    ...

</web-app>
```

2.8. Provider and Handlers

Handlers can be configured with Provider endpoints in `sun-jaxws.xml` descriptor or by putting `@HandlerChain` on the `Provider` implementation.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
        xmlns:javaee="http://java.sun.com/xml/ns/javaee" version='2.0'>
    <endpoint name='AddNumbers'
            implementation='provider.server.AddNumbersImpl'
            wsdl='WEB-INF/wsdl/AddNumbers.wsdl'
            service='{http://duke.example.org}AddNumbersService'
            port='{http://duke.example.org}AddNumbersPort'
            url-pattern='/addnumbers'/>
    <javaee:handler-chain>
        <javaee:handler-chain-name>my handler</javaee:handler-chain-name>
        <javaee:handler>
            <javaee:handler-name>MyHandler</javaee:handler-name>
```

```
            <javaee:handler-class>provider.server.MyHandler
            </javaee:handler-class>
        </javaee:handler>
    </javaee:handler-chain>
</endpoints>
```

3. AsyncProvider

Web Service endpoints may choose to work at the XML message level by implementing the `Provider` interface. The related information about `Provider` endpoints is documented in Provider page. However `Provider` endpoints are synchronous i.e. they receive XML requests and they return XML responses synchronously in `invoke()` method. If the endpoint wants to spawn a thread to process the request, it would block the jax-ws runtime thread and has to manage all the low details synchronizing the threads when the response is available. Also blocking a thread doesn't really scale well especially when the underlying transport is capable of handling asynchronous request and responses. RI provides an implemention specific solution to this problem by introducing `AsyncProvider.` This is similar to `Provider` endpoints but the difference is that the endpoint implementations have to implement `AsyncProvider` interface.

3.1. `AsyncProvider` Example

The following example shows an `AsyncProvider` example that accesses the payload of the request.

```
@WebServiceProvider
public class AsyncProviderImpl implements AsyncProvider<Source> {

    public void invoke(Source source, AsyncProviderCallback cbak,
                       WebServiceContext ctxt) {

        // ...

    }

}
```

3.2. `AsyncProvider` sample

See a `samples/asyncprovider/Readme.txt` that illustrates `AsyncProvider` endpoints.

3.3. `AsyncService` sample

See another `samples/asyncservice/Readme.txt"` sample that illustrates `AsyncProvider` endpoint that uses asynchronous servlet as the transport to bring true asynchronity on the server-side. See New Asynchronous Servlet Transport in Eclipse Implementation of XML Web Services [https://community.oracle.com/blogs/ramapulavarthi/2010/08/18/new-asynchronous-servlet-transport-jax-ws-ri] for more details on this feature.

4. Dispatch

Web service client applications may choose to work at the XML message level by using the `Dispatch<T>` APIs. The `javax.xml.ws.Dispatch<T>` interface provides support for the dynamic invocation of service endpoint operations.

Four Message Exchange Protocols(MEP) are supported: request-response, one way, asynchronous polling, and callback. Each of these invocation MEPs are required with Jakarta XML Binding data bound

`java.lang.Object`, `javax.xml.transform.Source`, `javax.xml.soap.SOAPMessage` and `javax.activation.DataSource` object requests.

The `javax.xml.ws.Service` acts as a factory for the creation of `Dispatch<T>` instances. In addition, a `Dispatch<T>` instance is created in either `Service.Mode.PAYLOAD` or `Service.Mode.MESSAGE` modes. A `javax.xml.soap.SOAPMessage` request can only be used with a `Dispatch<T>` instance of `Service.Mode.MESSAGE` and using the SOAP Binding. A `javax.activation.DataSource` request can only be used with a `Dispatch<T>` instance of `Service.Mode.MESSAGE` and using the XML/HTTP Binding.

Note that the `Dispatch<T>` instance simply acts as a conduit for the request. No validation of the message is required to be performed by the implementation, though some may catch errors during request processing. It is up to the client program to supply well-formed XML requests.

4.1. Programming Model

4.1.1. Create a dynamic `Service`.

The `javax.xml.ws.Service` acts as a factory for the creation of a dynamic `Service`. When created for use with `Dispatch<T>` APIs the `Service` created can be either a `Service` that has knowledge of the binding information of a known `Service` or no knowledge of any specific `Service`.

That is, when the `Service` is created with a WSDL file the port(s) binding ID, QName, and endpoint address are known to the `Service`.

The methods to create a dynamic `Service` are shown here:

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

A `Dispatch<T>` instance must be bound to a specific port and endpoint before use. The service has an `addPort(QName portName, URI bindingID, String endpointAddress)` method that the client program can invoke for `Dispatch<T>` objects. Ports created using this method can only be used with `Dispatch<T>` instances.

If the `Service` has been created with WSDL binding information the the port need not be added as the `Dispatch<T>` instance will be created specific for the binding information provided in the supplied WSDL file.

Developers who have used web service applications in the past are familiar with the port `QName` and endpoint address parameters of this method. Eclipse Implementation of XML Web Services 2.3.3 supports three `Binding` URI's, that of the SOAP 1.1, the SOAP 1.2 and XML/HTTP Binding. For more information on SOAP 1.2 support please refer to the SOAP 1.2 documents. For the XML/HTTP binding please see chapter 11 of the Jakarta XML Web Services Specification.

The addition of the SOAP 1.1 port using the `Service` API is shown here:

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

SOAP 1.2 support has been implemented for `Dispatch`. This requires only one change in the programming model. The addition of the SOAP 1.2 port using the `Service` API is shown here:

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

XML/HTTP binding support has been implemented for `Dispatch`. The creation of the XML/HTTP port using the `Service` API is shown here:

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

4.1.2. Create a `Dispatch<T>` instance.

The `Dispatch<T>` object can be created using either of these two `Service` methods:

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

For a `javax.xml.transform.Source` and Jakarta XML Binding data binding `java.lang.Object Dispatch<T>` can be created in both `Service.Mode.PAYLOAD` and `Service.Mode.MESSAGE` modes. A `javax.xml.soap.SOAPMessage` can only be created in `Service.Mode.MESSAGE` mode. The first form of the `createDispatch` method is used to create a `javax.xml.transform.Source` or `javax.xml.soap.SOAPMessage` specific to the `Dispatch<T>` instance.

A Jakarta XML Binding object-specific instance can only be created using the second method listed above.

It is important to note that once the `Dispatch<T>` instance is created it is static. That is, its `Service.Mode` or request type can not be changed. The instance can be reused given the caveat that if it is a Jakarta XML Binding-specific `Dispatch<T>` it must be reused with objects known to the same `JAXBContext`.

4.1.3. Set the context `Map<String, Object>` for the request.

The `Dispatch<T>` interface extends the `javax.xml.ws.BindingProvider` interface. The `BindingProvider` interface defines accessor methods for the request and response context maps. Standard `BindingProvider` properties are defined by the Jakarta XML Web Services specification and the client program may set and get these properties. The application may also define application-specific properties, but the specification discourages this for portability reasons.

4.1.4. Prepare the message request.

This is the client developer's responsibility. For examples of how to prepare specific request types refer to the `Dispatch<T>` sample applications.

4.1.5. Invoke the web service request.

Four types of invocation MEPs are supported using the methods below. In methods that produce a response, the type of `Object` returned will be of the same type as the request. For example, a `SOAPMessage` request will return a `SOAPMessage` response.

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

4.2. Dispatch and Asynchronous Invocations

Asynchronous invocations require special consideration. The first form of the `invokeAsync` method is a polling method. The response, `Response<T>`,returns to the user immediately and may be polled for completion. In the meantime, the client program can do other work.

The `javax.xml.ws.Response<T>` implements the `java.util.concurrent.Future<T>` interface that is included in Java SE. The `Response<T>` object returns the actual response via its `get` method, which blocks if the response is not ready to be returned.

The `Future<T>` interface also has a `cancel` method that will attempt to cancel the request invocation if the request is being invoked.

Faults returned from the service or exceptions thrown during the invocation are returned when the `Response<T>` `get` method is called. Because the execution doesn't occur in the main thread, the exception or fault returned is wrapped in an `java.util.concurrent.ExecutionException`. To obtain the actual cause use the `getCause` method of `ExecutionException`.

For more information on the `java.util.concurrent.Future<?>` interface see the Java SE documentation.

```
public interface Response<T> extends java.util.concurrent.Future<T> {

    Map<String, Object> getContext();

}
```

The second form of the `invokeAsync` method has a second parameter of type `javax.xml.ws.AsyncHandler`. The purpose of the `AsyncHandler` is to get and handle the the response or any fault thrown in an application-specific way. The `AsyncHandler` has a method `handleResponse(Response<T>)` that takes a `javax.xml.ws.Response<T>` parameter. This method gets the response or any faults and processes them according to behavior defined in the application. Note that it is the responsibility of the client program to implement the asynchronous handler.

```
class ResponseHandler implements javax.xml.ws.AsyncHandler {

    public handleResponse(Response<T>);

}
```

This form of the asynchronous invocation method returns a `Future<?>` object with wildcard type. As in the asynchronous poll invocation, the `Future<T>` object can be polled to see if the response is ready. However, calling the `get` method will not return the response of the invocation, but an object of indeterminate type.

Examples of synchronous and asynchronous invocations are shown in the `Dispatch<T>` samples. For convenience an example of `Response<T>` usage is display here:

```
Response<Source> response = dispatch.invokeAsync(Source);
while (!response.isDone()) {
    //go off and do some work
}

try {
    //get the actual result
    Source result = (javax.xml.transform.Source) response.get();
    //do something with the result
} catch (ExecutionException ex) {
    //get the actual cause
    Throwable cause = ex.getCause();
} catch (InterupptedException ie) {
    //note interruptions
    System.out.println("Operation invocation interrupted");
}
```

5. Asynchronous Client

This document describes how a client application can invoke a remote web service asynchronously. It can do so either by generating a static stub or using the Dispatch API.

5.1. Asynchronous Invocation Using Static Stub

Client application should apply `jaxws:enableAsyncMapping` binding declaration to the WSDL file to generate asynchronous method in the service endpoint interface. Please refer to Asynchrony for details on how this can be applied to the WSDL file.

Lets look at the following WSDL excerpt:

```
<definitions
        name="AddNumbers"
        targetNamespace="http://duke.example.org"
        xmlns:tns="http://duke.example.org"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

    ...

    <portType name="AddNumbersImpl">
        <operation name="addNumbers">
            <input message="tns:addNumbers"/>
            <output message="tns:addNumbersResponse"/>
        </operation>
    </portType>
    <binding name="AddNumbersImplBinding" 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>

    ...

</definitions>
```

In order to generate a service endpoint interface with asynchronous methods the following binding declaration file will be passed to **wsimport**:

```
<bindings
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        wsdlLocaption="http://localhost:8080/jaxws-async/addnumbers?WSDL"
        xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions">
        <package name="async.client"/>
        <enableAsyncMapping>true</enableAsyncMapping>
```

```
    </bindings>
</bindings>
```

It produces the following service endpoint interface (annotations are removed from the synchronous method for better readability):

```
//synchronous method
public int addNumbers(int number1, int number2) throws
 java.rmi.RemoteException;

//async polling Method
public Response<AddNumbersResponse> addNumbers(int number1, int number2);

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

5.1.1. Async Polling

```
//async polling Method
public Response<AddNumbersResponse> addNumbers(int number1, int number2);
```

Typically a client application will invoke the async polling operation on the stub and check for a response on the returned `Response` object. The response is available when `Response.isDone` returns true.

```
javax.xml.ws.Response<AddNumbersResponse> resp = port
        .addNumbersAsync(10, 20);
while (!resp.isDone()) {
    //do something
}
System.out.println("The sum is: " + resp.get().getReturn());

...
```

5.1.2. Async Callback

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

Here the client application provides an `AsyncHandler` by implementing the `javax.xml.ws.AsyncHandler<T>` interface.

```
/**
 * Async callback handler
 */
class AddNumbersCallbackHandler implements
        AsyncHandler<AddNumbersResponse> {
    private AddNumbersResponse output;

    /**
     * @see javax.xml.ws.AsyncHandler#handleResponse(javax.xml.ws.Response)
     */
    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 async handler is then passed as the last parameter of the async callback method:

```
//instantiates the callback handler
AddNumbersCallbackHandler callbackHandler = new
        AddNumbersCallbackHandler();

//invoke the async callback method
Future<?> resp = port.addNumbersAsync(number1, number2, callbackHandler);
while (!resp.isDone()) {
    //do something
}
System.out.println("The sum is: " +
 callbackHandler.getResponse().getReturn());
```

5.2. Asynchronous Invocation Using Dispatch

For information on the Dispatch API and asynchronous invocations see Dispatch

6. Handler

6.1. Handler Types

Jakarta XML Web Services defines a `Handler` interface, with subinterfaces `LogicalHandler` and `SOAPHandler`. The `Handler` interface contains `handleMessage(C context)` and `handleFault(C context)` methods, where `C` extends `MessageContext`. A property in the `MessageContext` object is used to determine if the message is inbound or outbound. `SOAPHandler` objects have access to the full soap message including headers. Logical handlers are independent of protocol and have access to the payload of the message.

The new handler types can now be written without casting the message context object that is passed to them. For instance:

```
public class MyLogicalHandler implements
        LogicalHandler<LogicalMessageContext> {

    public boolean handleMessage(LogicalMessageContext messageContext) {
        LogicalMessage msg = messageContext.getMessage();
        return true;
    }
    // other methods
}

public class MySOAPHandler implements SOAPHandler<SOAPMessageContext> {

    public boolean handleMessage(SOAPMessageContext messageContext) {
        SOAPMessage msg = messageContext.getMessage();
        return true;
    }
    // other methods
```

```
}
```

A `close(C context)` method has been added that is called on the handlers at the conclusion of a message exchange pattern. This allows handlers to clean up any resources that were used for the processing of a request-only or request/response exchange.

The `init()` and `destroy()` methods of the handler lifecycle no longer exist. Instead, a method may be annotated with the `@PostConstruct` annotation to be called after the handler is created or the `@PreDestroy` annotation to be called before the handler is destroyed. Note that the annotated methods must return `void` and take no arguments:

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class MyLogicalHandler implements
        LogicalHandler<LogicalMessageContext> {

    @PostConstruct
    public void methodA() {
    }

    @PreDestroy
    public void methodB() {
    }

    // other methods
}
```

6.2. MessageContext

In the examples above, the `LogicalMessage` object allows a handler to get and set the message payload either as a Jakarta XML Binding object or as a `javax.xml.transform.Source`. The `SOAPMessage` object allows access to headers and the SOAP body of the message.

Both context objects extend `MessageContext`, which holds properties that the handlers can use to communicate with each other. A standard property `MessageContext.MESSAGE_OUTBOUND_PROPERTY` holds a `Boolean` that is used to determine the direction of a message. For example, during a request, the property would be `Boolean.TRUE` when seen by a client handler and `Boolean.FALSE` when seen by a server handler.

The message context object can also hold properties set by the client or provider. For instance, port proxy and dispatch objects both extend `BindingProvider`. A message context object can be obtained from both to represent the request or response context. Properties set in the request context can be read by the handlers, and the handlers may set properties on the message context objects passed to them. If these properties are set with the scope `MessageContext.Scope.APPLICATION` then they will be available in the response context to the client. On the server end, a context object is passed into the `invoke` method of a `Provider`.

6.3. Handler Chain Configuration

6.3.1. Handler Files

Starting from a WSDL file, handler chain configuration is through WSDL customizations as defined by Jakarta Enterprise Web Services [https://jakarta.ee/specifications/enterprise-ws/]. A `<handler-chains>` element is added to the customization file. The following is a simple handler chain with one handler (customization may be on server or client side):

```
<-- excerpt from customization file -->
<bindings xmlns="http://java.sun.com/xml/ns/jaxws">
    <handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
        <handler-chain>
            <handler>
                <handler-class>fromwsdl.handler_simple.common.TestHandler
                </handler-class>
            </handler>
        </handler-chain>
    </handler-chains>
</bindings>
```

Multiple `handler-chain` elements may exist within the `handler-chains` element. These may optionally use a service name, port name, or protocol pattern in their description to apply some chains to certain ports and protocols and not to others. For instance (note the wildcard character used in the service name):

```
<-- excerpt -->
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
    <handler-chain>
        <service-name-pattern xmlns:ns1="urn:namespace">ns1:My*Service
        </service-name-pattern>
        <handler>...</handler>
    </handler-chain>

    <handler-chain>
        <port-name-pattern xmlns:ns1="urn:namespace">ns1:HelloPort
        </port-name-pattern>
        <handler>...</handler>
    </handler-chain>

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

Handlers will appear in the final handler chain in the order that they are included in the customization file. However, logical handlers will be sorted out and called before protocol handlers during execution.

Starting from a Java class, annotations are used to describe the handler chain as defined by Jakarta Web Services Metadata [https://jakarta.ee/specifications/web-services-metadata/]. The following example uses the `@HandlerChain` annotation to refer to a file describing the chain.

```
import javax.jws.HandlerChain;
import javax.jws.WebService;

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

An example `handlers.xml` file is shown below. The schema is the same that is used for the customization.

```
<?xml version="1.0" encoding="UTF-8"?>
<jws:handler-chains xmlns:jws="http://java.sun.com/xml/ns/javaee">
    <jws:handler-chain>
```

```
        <jws:handler>
            <jws:handler-class>fromjava.handler_simple.common.TestHandler
            </jws:handler-class>
        </jws:handler>
    </jws:handler-chain>
</jws:handler-chains>
```

When packaging the service, the `handlers.xml` file must be in the classpath within the WAR file, either directly under `WEB-INF/classes` or further down in the same package as the service class file.

On the server side, the handlers may be configured in the `sun-jaxws.xml`deployment descriptor as well. A handler chain specified here will override handlers in WSDL customizations or annotated classes. The schema for the handler section is the same as in the previous examples:

```
<endpoints ...>
    <endpoint...>
        <handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
            <handler-chain>
                ...
            </handler-chain>
        </handler-chains>
    </endpoint>
</endpoints>
```

6.3.2. Programmatic Case

Handler chains may be configured on the client side at runtime by setting a chain directly on a `Binding-Provider` (e.g., a `Dispatch` object or a port proxy) or by using a `HandlerResolver`. This example shows how to add a handler chain to a port proxy:

```
// given proxy interface HelloPortType
HelloPortType myProxy = // create proxy

Binding binding = ((BindingProvider) myProxy).getBinding();

// can create new list or use existing one
List<Handler> handlerList = binding.getHandlerChain();

handlerList.add(new MyHandler());
binding.setHandlerChain(handlerList);
```

To configure the handlers that are added to newly created `Binding` objects, add a handler resolver to the service with `setHandlerResolver()`. The new resolver will be used whenever a `Binding-Provider` is created from the service. An example resolver is as follows:

```
/*
 * Add handlers to the returned list based on the information
 * in info.getBindingID(), getPortName(), and/or getServiceName().
 */
public class MyResolver implements HandlerResolver {

    public List<Handler> getHandlerChain(PortInfo info) {
        List<Handler> handlers = new ArrayList<Handler>();
        // add handlers to list based on PortInfo information
        return handlers;
    }

}
```

A resolver that modifies the initially configured handler chains could be written by calling `service.getHandlerResolver()` and passing the original resolver to a new one:

```
// original HandlerResolver passed in constructor or setter method
public List<Handler> getHandlerChain(PortInfo info) {
    List<Handler> handlers = originalResolver.getHandlerChain(info);
    // alter list based on PortInfo information
    return handlers;
}
```

6.4. Handler Samples

The fromjavahandler and fromwsdlhandler samples set a `SOAPHandler` on the client and server. This handler simply outputs the contents of the SOAP message and can be used to see the requests and responses being passed back and forth. See the sample documentation for information on running the samples.

7. MTOM and Swaref

7.1. MTOM and XOP

MTOM [http://www.w3.org/TR/soap12-mtom/] (Message Transmission and Optimization Mechanism) together with XOP [http://www.w3.org/TR/xop10/] (XML Binary Optimized Packaging) defines how an XML binary data such as `xs:base64Binary` or `xs:hexBinary` can be optimally transmitted over the wire. XML type, such as `xs:base64Binary` is sent in lined inside the SOAP envelope. This gets quite in-efficient when the data size is more, for example a SOAP endpoint that exchanges images/songs etc. MTOM specifies how XOP packaging can be used to send the binary data optimally.

7.2. MTOM in Jakarta XML Web Services

MTOM feature is disabled in Jakarta XML Web Services by default. It can be enabled on the client and server. Once enabled all the XML binary data, XML elements of type `xs:base64Binary` and `xs:hexBinary` is optimally transmitted. Currently MTOM works only with proxy port.

Note: MTOM optimization is applied right at the time when Jakarta XML Web Services writes the message on to the wire. This is done to avoid any bufferring. Having a handler means that Jakarta XML Web Services transforms the Message in to DOM or SOAPMessage. It results in inlined base64 encoded data and it remains so when the data is written over the wire as attachment. This is done in order to avoid unnecessary conversion to and from when handlers are being used. In short: when handlers are used, MTOM optimization does not happen.

7.2.1. `xmime:expectedContentType` to Java type mapping

An schema element of type `xs:bas64Binary` or `xs:hexBinary` can be annotated by using attribute reference using `xmime:expectedContentType` [http://www.w3.org/TR/xml-media-types/] Jakarta XML Binding specification defines `xmime:expectedContentType` to Java type mapping in Table 2, "`xmime:expectedContentType` to Java type mapping". Here is this table:

Table 2. `xmime:expectedContentType` to Java type mapping

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

MIME Type	Java Type
text/plain	java.lang.String
text/xml or application/xml	javax.xml.transform.Source
/	javax.activation.DataHandler

```
<element name="image" type="base64Binary"/>
```

is mapped to `byte[]`

```
<element name="image" type="base64Binary"
        xmime:expectedContentTypes="image/jpeg"
        xmlns:xmime="http://www.w3.org/2005/05/xmlmime"/>
```

is mapped to `java.awt.Image`

7.2.2. `xmime:contentType` schema annotation

`xmime:contentType` [http://www.w3.org/TR/xml-media-types/] schema annotation indicates the content-type of an XML element content whose type is `xs:base64Binary` or `xs:hexBinary`. The value of the attribute is a valid `content-type` string (e.g., "`text/xml; charset=utf-16`"). This attribute specifies the `content-type` of the element content on which it occurs. This annotation can be primarily used to indicate the `Content-Type` of binary data.

For example the schema type

```
<element name="TestMtomXmimeContentType" type="types:PictureType"/>
<complexType name="PictureType">
    <simpleContent>
        <restriction base="xmime:base64Binary">
            <attribute ref="xmime:contentType" use="required"/>
        </restriction>
    </simpleContent>
</complexType>
```

Here `xmime:base64Binary` is defined by Describing Media Content of Binary Data in XML [http://www.w3.org/TR/xml-media-types/#schema].

Gets mapped to PicutreType bean by **wsimport**:

```
 PictureType req = new PictureType();
req.setValue(name.getBytes());
req.setContentType("application/xml");
```

On the wire this is how it looks:

```
 <?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv=" http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd=" http://www.w3.org/2001/XMLSchema"
        xmlns:ns1=" http://example.org/mtom/data"
        xmlns:ns2="http://www.w3.org/2005/05/xmlmime">
    <soapenv:Body>
        <ns1:TestMtomXmimeContentTypeResponse
                ns2:contentType="application/xml">
            <xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
                    href="c id:193ed174-
d313-4325-8eed-16cc25595e4e@example.org"/>
        </ns1:TestMtomXmimeContentTypeResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

7.2.3. How to enable MTOM in Jakarta XML Web Services

Enabling MTOM on Server:

- Enable using `@javax.xml.ws.soap.MTOM` annotation on the endpoint (SEI) implementation class

```
@javax.xml.ws.soap.MTOM
@WebService(endpointInterface = "mtom.server.Hello")
public class HelloImpl implements Hello {
    // ...
}
```

- MTOM can be also be enabled on an endpoint by specifying enable-mtom attribute to true on an endpoint element in sun-jaxws.xml deployment descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
        version='2.0'>
    <endpoint name="Mtom"
            implementation="mtom.server.HelloImpl"
            url-pattern="/hello"
            enable-mtom="true"/>
</endpoints>
```

- Enable using `@BindingType` on the endpoint (SEI) implementation class
  - `@BindingType(value=javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING)` will enable MTOM on the deployed endpoint for SOAP 1.1 binding
  - `@BindingType(value=javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_MTOM_BINDING)` will enable MTOM on the deployed endpoint for SOAP 1.2 binding

Enabling MTOM on Client:

- To enable MTOM on client-side, pass javax.xml.ws.soap.MTOMFeature as WebServiceFeature parameter while crating the Proxy or Dispatch. Here is the code snippet from the client `samples/mtom/src/mtom/client/MtomApp.java` of the mtom sample:
  - Hello port = new HelloService().getHelloPort(new MTOMFeature()); gives a proxy with MTOM enabled
  - javax.xml.ws.Service.createDispatch(....,new javax.xml.ws.soap.MTOMFeature()) gives a Dispatch instance with MTOM enabled
- Jakarta XML Web Services specification has defined API to enable and to check if the MTOM is enabled.
  - javax.xml.ws.soap.SOAPBinding.setMTOMEnabled(boolean enable) - enable or disable MTOM.
  - javax.xml.ws.soap.SOAPBinding.isMTOMEnabled() - returns true if MTOM is enabled otherwise false.

```
Hello port = new HelloService.getHelloPort();
//get the binding and enable mtom
SOAPBinding binding = (SOAPBinding) ((BindingProvider) port).getBinding();
boolean mtomEnabled = binding.isMTOMEnabled();

binding.setMTOMEnabled(true);
```

7.2.4. Attach vs In-line

As defined by Jakarta XML Binding specification `xs:base64Binary` and `xs:hexBinary` mapping to java is `byte[]`. Eclipse Implementation of XML Web Services has set a threshold of 1KB of `byte[]` size. This threshold can be modified using implementation specific property `com.sun.xml.ws.developer.JAXWSProperties.MTOM_THRESHOLD_VALUE` in the `RequestContext` on the client side and in the `MessageContext` on the server side. If the `byte[]` that is being sent is less than this threshold (default is 1KB) then the binary data is base64 encoded by Jakarta XML Binding and in lined inside the SOAP Body otherwise the binary data is sent as attachment mime part in Multipart/Related package and XML infoset for the binary data is XOP encoded by Jakarta XML Binding

```
<xop:Include href=...>
```

is used to reference the attachment. The XOP encoding and packaging is done as per described by the XOP packaging rules [http://www.w3.org/TR/xop10/#xop_packages]. The href is the the `Content-ID` of the attachment and is encoded as per CID URI scheme defined in RFC 2111 [http://www.faqs.org/rfcs/rfc2111.html]. `xmime:contentType` attribute may appear on the element that includes binary data to indicate preferred media type as annotated on the corresponding schema.

7.2.4.1. How to specify MTOM data Threshold

Default threshold value for MTOM feature is 0 bytes. You can set a threshold value for MTOM by using `@MTOM` annotation on server-side or using MTOMFeature on client-side. Let's say you set MTOM threshold as 3000, this serves as hint to Jakarta XML Web Services when to send binary dat as attachments. In this case, Jakarta XML Web Services will send any byte array in the message thats equal to or larger than 3KB as attachment and the corresponding XML infoset will be XOP encoded (will contain reference to this attachment)

- On Server-side, `@MTOM(threshold=3000)`
- On Client-side, pass `MTOMFeature(3000)` as `WebServiceFeature` as mentioned in Section 7.2.3, “How to enable MTOM in Jakarta XML Web Services”, while creating the proxy/dispatch.

7.2.5. MTOM Samples

Example 1. MTOM Sample - mtom

This is SOAP 1.1 MTOM SampleThis is how the Jakarta XML Web Services generated XOP packaged SOAP message looks on the wire:

```
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"/>
            </image>
        </Detail>
    </soapenv:Body>
</soapenv:Envelope>

------=_Part_1_4558657.1118953559446
Content-Type: image/jpeg
Content-ID: <5aeaa450-17f0-4484-b845-a8480c363444@example.org>

 #
    # #JFIF ## # # # ##
#¶
########¶########
    $.' ",###(7),01444#'9=82<.342 # C# #####2!#!
22222222222222222222222222222222222222222222

222222 # ) ¬##" ###### # # ########
    #### # # ## ############
    #}### ####!1A##Qa"q¶2?#B##§R##$3bré

#####
%&'()*456789:CDEFGHIJSTUVWXYZcdefghijstuvwxyzâäàåçêëèÆôöòûùÿÖÜóúñÑªº¿#¬####################
±####÷#°··
    #
```

Above Photo is inlined binary data because its less than 1KB and image which is more than 1KB is sent as attachment. Here is the associated schema:

```
<element name="Detail" type="types:DetailType"/>
<complexType name="DetailType">
    <sequence>
        <!-- mapped to byte[] -->
        <element name="Photo" type="base64Binary"/>
        <!-- mapped tojava.awt.Image -->
        <element name="image" type="base64Binary"
                 xmime:expectedContentTypes="image/jpeg"/>
    </sequence>
</complexType>
```

Example 2. MTOM Sample - mtom-soap12

This is SOAP 1.2 MTOM Sample. Here is how the Jakarta XML Web Services generated soap message looks on the wire:

```
<element name="image" type="base64Binary" xmime:expectedContentTypes="image/
jpeg"/>

Content-Type:
    Multipart/Related; start-info="application/soap+xml"; type="application/
xop+xml";
    boundary="----=_Part_0_1744155.1118960238280"
Content-Length: 1946
SOAPAction: ""

------=_Part_1_4558657.1118960238320
Content-Type: application/xop+xml; type="application/soap+xml"; charset=utf-8

<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/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:42a7ee0a-20ee-426b-
a135-094d72bc138f@example.org"/>
            </image>
        </Detail>
    </soapenv:Body>
</soapenv:Envelope>

------=_Part_1_4558657.1118960238320
Content-Type: application/octet-stream
Content-ID: <42a7ee0a-20ee-426b-a135-094d72bc138f@example.org>

 # # #JFIF ## # # # ##
#¶

    ########¶########
    $.' ",###(7),01444#'9=82<.342 # C# #####2!#!
22222222222222222222222222222222222222222222

222222 # ' )##" ###### # # ########
    #### # # ## ############
    #}### ####!1A##Qa"q¶2?#B##§R##$3bré

#####
%&'()*456789:CDEFGHIJSTUVWXYZcdefghijstuvwxyzâäàåçêëèÆôöòûùÿÖÜóúñÑªº¿#¬####################
±####÷#°··
    #
 ## ########## ####
```

7.3. swaRef

WS-I Attachment Profile 1.0 [http://www.ws-i.org/Profiles/AttachmentsProfile-1.0-2004-08-24.html] defines mechanism to reference MIME attachment parts using swaRef [http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html#Example_Attachment_Description_Using_swaRef]. In this mechanism the content of XML element of type `wsi:swaRef` [http://ws-i.org/profiles/basic/1.1/xsd/] is sent as MIME attachment and the element inside SOAP Body holds the reference to this attachment in the CID URI scheme as defined by RFC 2111 [http://www.faqs.org/rfcs/rfc2111.html].

7.3.1. swaRef in Jakarta XML Web Services

Jakarta XML Binding defines mapping of `wsi:swaRef` schema type to javax.activation.DataHandler. An application will construct the DataHandler with the data and the appropriate MIME type and Jakarta XML Web Services will coordinate with Jakarta XML Binding and Jakarta SOAP with Attachments to send it as attachment MIME part.

7.3.2. How to use swaRef

An XML element of type `wsi:swaRef` is mapped to a DataHandler and is sent as attachment over the wire. For example,

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

will be sent over the wire as :

```
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="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocaption="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/application_1_4.xsd" version="1.4">
    <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>
```

7.3.3. swaRef Sample

Refer to swaRef sample `testSwaRef()` method in `samples/mime/src/mime/client/MimeApp.java`

8. SOAP 1.2

8.1. Introduction

The default binding supported by Eclipse Implementation of XML Web Services is SOAP 1.1 over HTTP. With this release we have added **SOAP 1.2 binding over HTTP** support into Eclipse Implementation of XML Web Services. This document describes how SOAP 1.2 binding can be applied to an endpoint

and how it can be used on the client side in the case of proxy port. To enable SOAP 1.2 support in the `Dispatch` client please refer to the `Dispatch` documents.

8.2. SOAP 1.2 Endpoint

To enable SOAP 1.2 binding on an endpoint. You would need to set binding attribute value in sun-jaxws.xml to SOAP 1.2 HTTP binding value as specified by `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_BINDING` which is: "http://www.w3.org/2003/05/soap/bindings/HTTP/" or "http://java.sun.com/xml/ns/jaxws/2003/05/soap/bindings/HTTP/"

Here is the `sun-jaxws.xml` from fromjava-soap1.2 sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
        version="2.0">
    <endpoint name="fromjava-soap12"
            implementation="fromjava_soap12.server.AddNumbersImpl"
            binding="http://java.sun.com/xml/ns/jaxws/2003/05/soap/bindings/
HTTP/
            url-pattern="/addnumbers"/>
</endpoints>
```

Eclipse Implementation of XML Web Services generates WSDL on the fly when requested by a client. If this binding attribute is present and is equal to SOAP 1.2 HTTP binding WSDL with SOAP 1.2 binding is generated. Based on this binding descriptor Eclipse Implementation of XML Web Services runtime configures itself to handle SOAP 1.2 messages.

Notice that the binding id "http://java.sun.com/xml/ns/jaxws/2003/05/soap/bindings/HTTP/" is not a standard binding id. If you use SOAP 1.2 binding id "http://www.w3.org/2003/05/soap/bindings/HTTP/" defined by Jakarta XML Web Services, still the endpoint is configured to use SOAP 1.2 binding, except that a wsdl will not be generated on the fly.

Alternatively, you can specify the binding through `@BindingType` annotation in the implementation class to use SOAP 1.2 binding. Here is an example from the fromjava_soap12 sample.

```
@WebService
@BindingType(value = "http://java.sun.com/xml/ns/jaxws/2003/05/soap/bindings/
HTTP/")
public class AddNumbersImpl {

    /**
     * @param number1
     * @param number2
     * @return The sum
     * @throws AddNumbersException if any of the numbers to be added is
     *                             negative.
     */
    public int addNumbers(int number1, int number2) throws
            AddNumbersException {
        if (number1 < 0 || number2 < 0) {
            throw new AddNumbersException("Negative number cant be added " +
                    "!", "Numbers: " + number1 + ", " + number2);
        }
        return number1 + number2;
    }

}
```

The commandline **wsgen** and the equivalent ant task can be used to generate SOAP 1.1 (default) or SOAP 1.2 WSDL. The binding information should be passed using `-wsdl:protocol` switch.

8.3. Client

On the client there is nothing special that has to be done. Eclipse Implementation of XML Web Services runtime looks into the WSDL to determine the binding being used and configures itself accordingly. **wsimport** command line tool or **wsimport** ant task can be used to import the WSDL and to generated the client side artifacts.

8.4. Samples

There are 2 samples bundled with this release

- fromwsdl-soap12 - shows SOAP 1.2 endpoint developed starting from wsdl
- fromjava-soap12 - shows SOAP 1.2 endpoint developed starting from Java

A SOAP 1.2 message generated by Eclipse Implementation of XML Web Services:

```
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 178
SOAPAction: ""

<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
    <soapenv:Body>
        <addNumbers xmlns="http://duke.example.org">
            <arg0>-10</arg0>
            <arg1>20</arg1>
        </addNumbers>
    </soapenv:Body>
</soapenv:Envelope>
```

A SOAP 1.2 Fault message generated by Eclipse Implementation of XML Web Services:

```
Content-Type:application/soap+xml; charset=utf-8
Content-Length: 476
SOAPAction: ""

<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
    <soapenv:Body>
        <soapenv:Fault>
            <soapenv:Code>
                <soapenv:Value>
                    soapenv:Sender
                </soapenv:Value>
            </soapenv:Code>
            <soapenv:Reason>
                <soapenv:Text xml:lang="en">
                    Negative number cant be added!
                </soapenv:Text>
            </soapenv:Reason>
            <soapenv:Detail>
                <AddNumbersFault xmlns="http://duke.example.org">
                    <faultInfo>Numbers: -10, 20</faultInfo>
                    <message>Negative number cant be added!</message>
                </AddNumbersFault>
            </soapenv:Detail>
```

```
        </soapenv:Fault>
    </soapenv:Body>
</soapenv:Envelope>
```

9. WSDL Customization

The Jakarta XML Web Services specification defines standard XML-based customization for WSDL to Java mapping and to control certain features. These customizations, or *binding declarations*, can customize almost all WSDL components that can be mapped to Java, such as the service endpoint interface class, method name, parameter name, exception class, etc. The other important thing you can do with these binding declarations is control certain features, such as asynchrony, provider, wrapper style, and additional headers. For example, a client application can enable asynchrony for a particular operation in a `portType` or all operations in a `portType` or all `portType` operations defined in the WSDL file.

The JAX-RPC 1.1 specification did not define a standard customization architecture. However JAX-RPC 1.x SI had limited WSDL to Java customization support. It allowed a JAX-RPC 1.x application to:

- Define a package where Java artifacts mapped from a WSDL file will be generated.
- Customize the package for the value classes mapped from the imported XML schema by the WSDL document.
- Customize handler chains.

But these customizations were not portable and could not be used across other JAX-RPC implementations. Eclipse Implementation of XML Web Services 2.3.3 provides complete support for all the binding declarations defined by the specification.

9.1. Declaring Customizations

All the binding declaration elements live in `http://java.sun.com/xml/ns/jaxws` namespace. There are two ways to specify binding declarations. In the first approach, all binding declarations pertaining to a given WSDL document are grouped together in a standalone document, called an *external binding file*. The second approach consists of embedding binding declarations directly inside a WSDL document. In either case, the `jaxws:bindings` element is used as a container for Jakarta XML Web Services binding declarations. The `jaxws` prefix maps to the `http://java.sun.com/xml/ns/jaxws` namespace.

9.1.1. External Binding Declaration

External binding files are semantically equivalent to embedded binding declarations. When **wsimport** processes the WSDL document for which there is an external binding file, it internalizes the binding declarations defined in the external binding file on the nodes in the WSDL document they target using the `wsdlLocation` attribute. The embedded binding declarations can exist in a WSDL file and an external binding file targeting that WSDL, but **wsimport** may give an error if, upon embedding the binding declarations defined in the external binding files, the resulting WSDL document contains conflicting binding declarations.

9.1.1.1. Root Binding Element

The `jaxws:bindings` declaration appears as the root of all other binding declarations. This top-level `jaxws:bindings` element must specify the location of the WSDL file as a URI in the value of `wsdlLocation` attribute.

Its important that the `wsdlLocation` attribute on the root `jaxws:bindings` declaration is same as the WSDL location URI given to **wsimport**.

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

    ...

</jaxws:bindings>
```

9.1.1.2. Child Binding Elements

The root `jaxws:bindings` element may contain child `jaxws:bindings` elements. In this case the child `jaxws:bindings` element must carry an XPath expression in the node attribute to refer to the WSDL node it customizes.

Here is an excerpt from an external binding file `samples/external-customize/etc/custom-client.xml` in the `external-customize` sample:

```
<jaxws:bindings
        wsdlLocation="http://localhost:8080/jaxws-external-customize/
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="external_customize.client"/>

        ...

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

In this example the child `jaxws:bindings` applies package customization. An XPath expression in the node attribute refers to the root node of the WSDL document, which is `wsdl:definitions` and declares the package `external_customize.client` for all the generated Java classes mapped from the WSDL file.

9.1.2. Embedded Binding Declarations

Embedded binding declarations directly inside the WSDL follow different rules compared to the binding declarations declared in the external binding file. Here are some important facts and rules as defined in the Jakarta XML Web Services specification:

- An embedded binding declaration is specified by using the `jaxws:bindings` element as a WSDL extension inside the wsdl node that is to be customized.
- When a `jaxws:bindings` element is used as a WSDL extension, the `jaxws:bindings` element should not have node attribute (the node attribute is only used in external customization file to scope the custmization to a wsdl element).
- A binding declaration embedded in a WSDL can only affect the WSDL element it extends.

Here's an example of embedded binding declarations in the WSDL `AddNumbers.wsdl` from the `inline-customize` sample:

```
<wsdl:portType name="AddNumbersImpl">
    <!-- wsdl:portType customizations -->
    <jaxws:bindings xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
        <!-- rename the generated SEI from AddNumbersImpl to MathUtil -->
```

```
        <jaxws:class name="MathUtil"/>
        ...
    </jaxws:bindings>
    <wsdl:operation name="addNumber">
        ...
    </wsdl:operation>
</wsdl:portType>
```

The above WSDL file excerpt shows the `wsdl:portType` customization. `jaxws:bindings` appears as extension element of `portType`. It customizes the class name of the generated service endpoint interface. Without this customization, or by default, the service endpoint interface class is named after the `wsdl:portType` name. The binding declaration `jaxws:class` customizes the generated class to be named `MathUtil` instead of `AddNumberImpl`.

In the following section, all the possible standard customizations and their scope is described. Global customizations can be specified under <wsdl:definitions> element and other customizations can be specified under the node of its scope.

9.2. Standard Customizations

This section provides the details of all the possible WSDL binding declarations.

9.2.1. Global bindings

The global customizations are the customizations that applies to the entire scope of `wsdl:definition` in the wsdl referenced by the root `jaxws:bindings@wsdlLocation.Following` customizations have the global scopes:

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

<jaxws:enableWrapperStyle/>

<jaxws:enableAsyncMapping/>
```

These can appear as direct child of the Root Binding Element declarations in the external customization file. For example:

```
<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">

    <package name="external_customize.client"/>
    <enableWrapperStyle>true</enableWrapperStyle>
    <enableAsyncMapping>false</enableAsyncMapping>
</bindings>
```

In embedded usage, the global customization can be specified under `<wsdl:definitions>` node of the wsdl.

9.2.2. Package Customization

By default **wscompile** generates WSDL artifacts in a package computed from the WSDL `targetNamespace`. For example, a WSDL file with the `targetNamespace http://duke.example.org` without any package customization will be mapped to the `org.duke` package. To customize the default package mapping you would use a `jaxws:package` customization on the `wsdl:definitions` node or it can directly appear inside the top level bindings element.

An important thing to note is that `-p` option on commandline **wsimport.sh** tool (package attribute on **wsimport** ant task), overrides the `jaxws:package` customization,it also overrides the schema package customization specified using jaxb schema customization.

For example:

```
<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">
    <package name="external_customize.client">
        <javadoc>Mathutil package</javadoc>
    </package>

    ...

</bindings>
```

or

```
<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="external_customize.client">
            <javadoc>Mathutil package</javadoc>
        </package>

        ...

    </bindings>

    ...

</bindings>
```

9.2.3. Wrapper Style

wsimport by default applies wrapper style rules to the abstract operation defined in the `wsdl:portType`, and if an operation qualifies the Java method signature is generated accordingly. Wrapper style Java method generation can be disabled by using `jaxws:enableWrapperStyle`.

`jaxws:enableWrapperStyle` can appear on the toplevel bindings element (with `@wsdlLocation` attribute), it can also appear on the following target nodes:

- `wsdl:definitions` global scope, applies to all the `wsdl:operations` of all `wsdl:portType` attributes
- `wsdl:portType` applies to all the `wsdl:operations` in the `portType`
- `wsdl:operation` applies to only this `wsdl:operation`

For example:

```
<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">
```

```
    <!-- applies to wsdl:definitions node, that would mean the entire wsdl --
>
    <enableWrapperStyle>true</enableWrapperStyle>
    <!-- wsdl:portType operation customization -->
    <bindings
            node="wsdl:definitions/wsdl:portType[@name='AddNumbersImpl']/
wsdl:operation[@name='addNumbers']">
        <!-- change java method name from addNumbers() to add() -->
        <enableWrapperStyle>false</enableWrapperStyle>

        ...

    </bindings>

    ...

</bindings>
```

In the example above the wrapper style is disabled for the `addNumbers` operation in `AddNumbersImpl portType` .This is because **wsimport** processes this binding in the following order: first `wsdl:operation`, then its parent `wsdl:portType`, and finally `wsdl:definitions`. Here `wsdl:operation addNumbers` has this customization disabled so this is what is applied by **wsimport** to generate a bare Java method signature.

9.2.4. Asynchrony

A client application can use the `jaxws:enableAsyncMapping` binding declaration so that **wsimport** will generate async polling and callback operations along with the normal synchronous method when it compiles a WSDL file.

It has the same target nodes as the wrapper style binding declaration described above in section 2.2.

- `wsdl:definitions` or toplevel bindings element: global scope, applies to all the `wsdl:operations` of all `wsdl:portType`
- `wsdl:portType` applies to all the `wsdl:operations` in the `portType`
- `wsdl:operation` applies to only this `wsdl:operation`

Example :

```
<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">

    <!-- applies to wsdl:definitions node, that would mean the entire wsdl --
>
    <enableAsyncMapping>false</enableAsyncMapping>
    <!-- wsdl:portType operation customization -->
    <bindings
            node="wsdl:definitions/wsdl:portType[@name='AddNumbersImpl']/
wsdl:operation[@name='addNumbers']">
        <!-- change java method name from addNumbers() to add() -->
        <enableAsyncMapping>true</enableAsyncMapping>

        ...

    </bindings>
```

```
    ...

</bindings>
```

In the above example **wsimport** will generate async polling and callback methods for the `addNumbers` operation. In the `wsdl:definition` node, the async customization is disabled or false but the `wsdl:operation` node has it enabled or true, and so **wsimport** generates the async methods of the `wsdl:operation` `addNumbers`.

This is how the generated signatures look (annotations are removed from synchronous method for reading simplicity):

```
//synchronous method
public int addNumbers(int number1, int number2)
        throws org.duke.AddNumbersFault_Exception, java.rmi.RemoteException;

//async polling Method
public Response<AddNumbersResponse> addNumbers(int number1, int number2);

//async callback Method
public Future<?> addNumbers(int number1, int number2,
                            AsyncHandler<AddNumbersResponse>);

...
```

9.2.5. The Provider Interface

By default the value of `jaxws:provider` binding is false. That is, provider interface generation is disabled. In order to mark a port as provider interface this binding declaration should refer to the `wsdl:port` node using an XPath expression. Please note that provider binding declaration applies only when developing a server starting from a WSDL file.

9.2.6. Class Customization

The generated class for `wsdl:portType`, `wsdl:fault`, `soap:headerfault`, and `wsdl:server` can be customized using the `jaxws:class` binding declaration. Refer to the external binding declaration file `custom-client.xml` in the `external-customize` sample.

9.2.6.1. The Service Endpoint Interface Class

wscompile will generate the service endpoint interface class `MathUtil` instead of the default `AddNumbersImpl` in this example:

```
<!-- wsdl:portType customization -->
<bindings node="wsdl:definitions/wsdl:portType[@name='AddNumbersImpl']">
    <!-- change the generated SEI class -->
    <class name="MathUtil">
        <javadoc>Perform mathematical computations</javadoc>
    </class>
</bindings>
```

9.2.6.2. The Exception Class

wsimport will generate the `MathUtilException` class instead of the default `AddNumbersException` in this example:

```
<!-- change the generated exception class name -->
<bindings
```

```
        node="wsdl:definitions/wsdl:portType[@name='AddNumbersImpl']/
wsdl:operation[@name='addNumbers']/wsdl:fault[@name='AddNumbersException']">
    <class name="MathUtilException">
        <javadoc>Exception generated during computation</javadoc>
    </class>
</bindings>
```

9.2.6.3. The Service Class

wsimport will generate `MathUtilService` instead of the default `AddNumbersService` in this example:

```
<!-- wsdl:service customization -->
<bindings node="wsdl:definitions/wsdl:service[@name='AddNumbersService']">
    <!-- change the generated service class -->
    <class name="MathUtilService">
        <javadoc>Service to perform mathematical computations</javadoc>
    </class>
</bindings>
```

9.2.7. Java Method Customization

The `jaxws:method` binding declaration is used to customize the generated Java method name of a service endpoint interface and to customize the port accessor method in the generated `Service` class. Refer to the external binding declaration file `custom-client.xml` in the `external-customize` sample.

9.2.7.1. Service Endpoint Interface Methods

wsimport will generate a method named `add` instead of the default `addNumbers` in this example:

```
<!-- wsdl:portType operation customization -->
<bindings
        node="wsdl:definitions/wsdl:portType[@name='AddNumbersImpl']/
wsdl:operation[@name='addNumbers']">
    <!-- change java method name from addNumbers() to add() -->
    <method name="add">
        <javadoc>Adds the numbers</javadoc>
    </method>
</bindings>
```

9.2.7.2. Port Accessor Methods in the Service Class

wsimport will generate the `getMathUtil` port accessor method in the generated `Service` class instead of the default `getAddNumbersImplPort` method in this example:

```
<!-- change the port accessor method -->
<bindings
        node="wsdl:definitions/wsdl:service[@name='AddNumbersService']/
wsdl:port[@name='AddNumbersImplPort']">
    <method name="getMathUtil">
        <javadoc>Returns MathUtil port</javadoc>
    </method>
</bindings>
```

9.2.8. Java Parameter Customization

The `jaxws:parameter` binding declaration is used to change the parameter name of generated Java methods. It can be used to change the method parameter of a `wsdl:operation` in a

`wsdl:portType`. Refer to the external binding declaration file `custom-client.xml` of the `external-customize` sample.

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

    ...

</bindings>
```

The above sample renames the default parameter name of the Java method `addNumbers` from `number1` to `num1`.

9.2.9. Javadoc customization

`jaxws:javadoc` customization can be used to specify javadoc text for java package, class(SEI, Service or Exception class) and on the methods in SEI and service class. Inorder to do it,it should appear on the corresponding wsdl nodes.

For package level javadoc:

```
<jaxws:package name="xs:string">
    <jaxws:javadoc>xs:string</jaxws:javadoc>
</jaxws:package>
```

For class level javadoc:

```
<jaxws:class name="xs:string">
    <jaxws:javadoc>xs:string</jaxws:javadoc>
</jaxws:class>
```

For method level javadoc:

```
<jaxws:method name="xs:string">
    <jaxws:javadoc>xs:string</jaxws:javadoc>
</jaxws:method>
```

For specific samples on javadoc customization for class, refer The Service Endpoint Interface Class, The Exception Class and The Service Class customization. For javadoc customization on method refer Service Endpoint Interface Methods and Port Accessor Methods in the Service Class customization and for package level customization refer Package Customization.

9.2.10. XML Schema Customization

An XML schema inlined inside a compiled WSDL file can be customized by using standard Jakarta XML Binding bindings. These Jakarta XML Binding bindings can live inside the schema or as the child of a `jaxws:bindings` element in an external binding declaration file:

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

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

External XML schema files imported by the WSDL file can be customized using a Jakarta XML Binding external binding declaration file:

```
<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:8080/jaxws-external-customize/
schema1.xsd"
            node="/xsd:schema">
        <jxb:schemaBindings>
            <jxb:package name="fromjava.client"/>
        </jxb:schemaBindings>
    </jxb:bindings>

    ...

</jxb:bindings>
```

The external Jakarta XML Binding binding declaration file can be passed to **wsimport** using the `-b` switch. See the Jakarta XML Web Services **wsimport** documentation for details.

9.2.11. Handler Chain Customization

`jaxws:bindings` customization can be used to customize or add handlers. All that is needed is to inline a handler chain configuration conforming to Jakarta Web Services Metadata Handler Chain configuration schema inside `jaxws:bindings` element.

Below is a sample Jakarta XML Web Services binding declaration file with JSR 181 handler chain configuration:

```
<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="http://java.sun.com/xml/ns/jaxws"
        xmlns:javaee="http://java.sun.com/xml/ns/javaee">
    <jaxws:bindings node="wsdl:definitions">
        <javaee:handler-chain>
            <javaee:handler-chain-name>LoggingHandlers
            </javaee:handler-chain-name>

            <javaee:handler>
                <javaee:handler-name>Logger</javaee:handler-name>
                <javaee:handler-class>fromwsdlhandler.common.LoggingHandler
                </javaee:handler-class>
            </javaee:handler>
        </javaee:handler-chain>
    </jaxws:bindings>
</jaxws:bindings>
```

When this customization file is passed on to **wsimport** tool using -b switch together with the WSDL, **wsimport** generates all the artifacts togather with a handler configuration file which has everything inside `jaxws:bindings` element enclosing the `jws:handler-chain` element. It also add `@javax.jws.HandlerChain` annotation in the generated SEI class. Eclipse Implementation of XML Web Services runtime uses the `@HandlerChain` annotation from the SEI to find the handlers that has to be added into the handle chain.

10. Annotations

10.1. Overview

Annotations play a critical role in Jakarta XML Web Services. First, annotations are used in mapping Java to WSDL and schema. Second, annotations are used a runtime to control how the Jakarta XML Web Services runtime processes and responds to web service invocations. Currently the annotations utilized by Jakarta XML Web Services are defined in separate JSRs:

- Jakarta Web Services Metadata 2.1 [https://jakarta.ee/specifications/web-services-metadata/]
- Jakarta XML Binding 2.3 [https://jakarta.ee/specifications/xml-binding/]
- Jakarta XML Web Services 2.3 [https://jakarta.ee/specifications/xml-web-services/]
- Jakarta Annotations [https://jakarta.ee/specifications/annotations/].

10.2. Jakarta Web Services Metadata Annotations

Because Jakarta Web Services Metadata has been written to work with JAX-RPC 1.1, we have made slight changes in the use and interpretation of these annotations to work better with Jakarta XML Web Services. We are working with the Jakarta Web Services Metadata expert group to align the next release with Jakarta XML Web Services and we hope that all of the changes we have made will be folded in.

10.2.1. `@javax.jws.WebService`

The purpose of this annotation is to mark an endpoint implementation as implementing a web service or to mark that a service endpoint interface as defining a web service interface. All endpoint implementation classes MUST have a `WebService` annotation and must meet the requirements of section 3.3 of the Jakarta XML Web Services [https://jakarta.ee/specifications/xml-web-services/] specification.

Table 3. `@javax.jws.WebService` - Description of Properties

Property	Description	Default
`name`	The name of the `wsdl:portType`	The unqualified name of the Java class or interface
`target-Namespace`	The XML namespace of the the WSDL and some of the XML elements generated from this web service. Most of the XML elements will be in the namespace according to the Jakarta XML Binding mapping rules.	The namespace mapped from the package name containing the web service according to section 3.2 of the Jakarta XML Web Services [https://jakarta.ee/specifications/xml-web-services/] specification.
`service-Name`	The Service name of the web service: `wsdl:service`	The unqualified name of the Java class or interface + "Service"
`end-pointIn-terface`	The qualified name of the service endpoint interface. If the implementation bean references a service endpoint interface, that service endpoint interface is used to determine the abstract WSDL contract (portType and bindings). In this case, the service implementation bean	None – If not specified, the endpoint implementation class is used to generate the web service contract. In this case,

Property	Description	Default
	MUST NOT include any Jakarta Web Services Metadata annotations other than `@WebService` and `@HandlerChain` In addition, the `@WebService` annotation MUST NOT include the name annotation element. The endpoint implementation class is not required to implement the endpointInterface.	a service endpoint interface is not required.
`portName`	The `wsdl:portName`	The `WebService.name` + "Port"
`wsdlLocation`	Not currently used by Eclipse Implementation of XML Web Services 2.3.3	

10.2.1.1. Annotation Type Definition

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface WebService {
    String name() default "";

    String targetNamespace() default "";

    String serviceName() default "";

    String wsdlLocation() default "";

    String endpointInterface() default "";

    String portName() default "";
}
```

10.2.1.2. Examples

Example 3. `@javax.jws.WebService` - Example 1

```
@WebService(name = "AddNumbers",
        targetNamespace = "http://duke.example.org")
public class AddNumbersImpl {
    /**
     * @param number1
     * @param number2
     * @return The sum
     * @throws AddNumbersException if any of the numbers to be added is
     *                             negative.
     */
    public int addNumbers(int number1, int number2) throws
            AddNumbersException {
        if (number1 < 0 || number2 < 0) {
            throw new AddNumbersException("Negative number cant be added " +
                    "!", "Numbers: " + number1 + ", " + number2);
        }
        return number1 + number2;
    }
}
```

If you are familiar with JAX-RPC 1.1, you will notice that the `AddNumbersImpl` implementation class does not implement a service endpoint interface. In Jakarta XML Web Services $ws.spec.version; a ser-

vice endpoint interface is no longer required. If a service endpoint interfaces is desired, then the `@WebService` annotation on the endpoint implementation is modified to specify the endpoint interface and the actual service endpoint interface must also have a `@WebService` annotation. The following is the above `AddNumbersImpl` modified to use a service endpoint interface.

Example 4. `@javax.jws.WebService` - Example 2 - Implementation class using Service Endpoint Interface

```
@WebService(endpointInterface = "annotations.server.AddNumbersIF")
public class AddNumbersImpl {
    /**
     * @param number1
     * @param number2
     * @return The sum
     * @throws AddNumbersException if any of the numbers to be added is
     * negative.
     */
    public int addNumbers(int number1, int number2) throws
            AddNumbersException {
        if (number1 < 0 || number2 < 0) {
            throw new AddNumbersException("Negative number cant be " +
                    "added!", "Numbers: " + number1 + ", " + number2);
        } return number1 + number2;
    }
}

@WebService(targetNamespace = "http://duke.example.org",
        name = "AddNumbers")
public interface AddNumbersIF extends Remote {

    public int addNumbers(int number1, int number2) throws
            RemoteException, AddNumbersException;

}
```

10.2.2. `@javax.jws.WebMethod`

The purpose of this annotation is to expose a method as a web service operation. The method must meet all the requirements of section 3.4 of the Jakarta XML Web Services [https://jakarta.ee/specifications/xml-web-services/] specification.

Table 4. `@javax.jws.WebMethod` - Description of Properties

Property	Description	Default
`operationName`	The name of the `wsdl:operation` matching this method. For operations using the mode defined by `SOAPBinding.Style.DOCUMENT`, `SOAPBinding.Use.LITERAL`, and `SOAPBinding.ParameterStyle.BARE`, this name is also used for the global XML element representing the operations body element. The namespace of this name is taken from the value `@WebService.targetNamespace` or its default value.	The name of the Java method
`action`	The XML namespace of the the WSDL and some of the XML elements generated from this web service. Most of	""

Property	Description	Default
	the XML elements will be in the namespace according to the Jakarta XML Binding mapping rules.	
`exclude`	Used to exclude a method from the WebService.	false

10.2.2.1. Annotation Type Definition

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface WebMethod {
    String operationName() default "";

    String action() default "";

    boolean exclude() default false;
}
```

10.2.2.2. Examples

Example 5. `@javax.jws.WebMethod` - Example

```
@WebService(targetNamespace = "http://duke.example.org",
        name = "AddNumbers")
public interface AddNumbersIF extends Remote {

    @WebMethod(operationName = "add", action = "urn:addNumbers")
    public int addNumbers(int number1, int number2) throws
            RemoteException, AddNumbersException;

}
```

10.2.3. `@javax.jws.OneWay`

The purpose of this annotation is to mark a method as a web service one-way operation. The method must meet all the requirements of section 3.4.1 of the Jakarta XML Web Services [https://jakarta.ee/specifications/xml-web-services/] spec.

There are no properties on the `OneWay` annotation.

10.2.3.1. Annotation Type Definition

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Oneway {
}
```

10.2.3.2. Examples

Example 6. `@javax.jws.OneWay` - Example

```
@WebService(name = "CheckIn")
public class CheckInIF {

    @WebMethod
    @OneWay
    public void checkIn(String name);

}
```

10.2.4. `@javax.jws.WebParam`

This annotation is used to customize the mapping of a single parameter to a message part or element.

Table 5. `@javax.jws.WebParam` - Description of Properties

Property	Description	Default
`name`	Name of the parameter. If the operation is RPC style and `@WebParam.partName` has not been specified, this is name of the `wsdl:part` representing the parameter. `@WebMethod.operation` Name, if the operation is document style and the parameter style is `BARE`. Otherwise, the default is `argN` if the operation is document style or the parameter maps to a header, this is the local name of the XML element representing the parameter. A name MUST be specified if the operation is document style, the parameter style is `BARE`, and the mode is `OUT` or `INOUT`.	`@WebMethod.operation` Name, if the operation is document style and the parameter style is `BARE`. Otherwise, the default is `argN`, where `N` represents the index of the parameter in the method signature (starting at `arg0`).
`target-Namespace`	The XML namespace for the parameter. Only used if the operation is document style or the paramater maps to a header. If the target namespace is set to "", this represents the empty namespace.	The empty namespace, if the operation is document style, the parameter style is `WRAPPED`, and the parameter does not map to a header. Otherwise, the default is the `targetNamespace` for the Web Service.
`mode`	Represents the direction the parameter flows for this method. Possible values are `IN`, `INOUT` and `OUT`. `INOUT` and `OUT` modes can only be used with parameters that meet the requirements for a holder as classified by section 3.5 of the Jakarta XML Web Services [https://jakarta.ee/specifications/xml-web-services/] specification. `OUT` and `INOUT` parameters can be used by all `RPC` and `DOCUMENT` bindings.	`IN` for non-holder parameters `INOUT` for holder parameters.
`header`	Specifies whether the parameter should be carried in a header.	`FALSE`
`partName`	Used to specify the `partName` for the parameter with `RPC` or `DOCUMENT`/`BARE` operations.	`@WebParam.name`

10.2.4.1. Annotation Type Definition

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target({ElementType.PARAMETER})
public @interface WebParam {

    public enum Mode {
        IN,
        OUT,
```

```
        INOUT
    }

    String name() default "";

    String targetNamespace() default "";

    Mode mode() default Mode.IN;

    boolean header() default false;

    String partName() default "";
}
```

10.2.4.2. Examples

Example 7. `@javax.jws.WebParam` - Example 1

```
@WebService(targetNamespace = "http://duke.example.org",
        name = "AddNumbers")
public interface AddNumbersIF extends Remote {

    @WebMethod(operationName = "add", action = "urn:addNumbers")
    @WebResult(name = "return")
    public int addNumbers(@WebParam(name = "num1") int number1,
                          @WebParam(name = "num2") int number2) throws
            RemoteException, AddNumbersException;

}
```

Example 8. `@javax.jws.WebParam` - Example 2

```
@WebService(targetNamespace = "http://duke.example.org",
        name = "AddNumbers")
public interface AddNumbersIF extends Remote {

    @WebMethod(operationName = "add", action = "urn:addNumbers")
    @WebResult(name = "return")
    public void addNumbers(@WebParam(name = "num1") int number1,
                           @WebParam(name = "num2") int number2,
                           @WebParam(name = "result",
            mode = WebParam.Mode.OUT) Holder<Integer> result) throws
 RemoteException, AddNumbersException;


}
```

10.2.5. `@javax.jws.WebResult`

This annotation is used to customize the mapping of the method return value to a WSDL part or XML element.

Table 6. `@javax.jws.WebResult` - Description of Properties

Property	Description	Default
`name`	The name of the return value in the WSDL and on the wire. For `RPC` bindings this is the part name of the return value in the response message. For `DOCUMENT` bindings	"return" for `RPC` and `DOCUMENT`/`WRAPPED` bindings. Method name + "Response"

Property	Description	Default
	this is the local name of the XML element representing the return value.	for DOCUMENT/BARE bindings.
targetNamespace	The XML namespace for the return value. Only used if the operation is document style or the return value maps to a header. If the target namespace is set to "", this represents the empty namespace.	The empty namespace, if the operation is document style, the parameter style is WRAPPED, and the return value does not map to a header, Otherwise, the default is the targetNamespace for the Web Service.
header	Specifies whether the result should be carried in a header.	FALSE
partName	Used to specify the partName for the result with RPC or DOCUMENT/BARE operations.	@WebResult.name

10.2.5.1. Annotation Type Definition

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface WebResult {

    String name() default "return";

    String targetNamespace() default "";

    boolean header() default false;

    String partName() default "";
}
```

10.2.5.2. Examples

Example 9. `@javax.jws.WebResult` - Example

```
@WebService(targetNamespace = "http://duke.example.org",
        name = "AddNumbers")
public interface AddNumbersIF extends Remote {

    @WebMethod(operationName = "add", action = "urn:addNumbers")
    @WebResult(name = "return")
    public int addNumbers(@WebParam(name = "num1") int number1,
                          @WebParam(name = "num2") int number2) throws
            RemoteException, AddNumbersException;

}
```

10.2.6. `@javax.jws.HandlerChain`

This annotation is used to specified an externally defined handler chain.

Table 7. `@javax.jws.HandlerChain` - Description of Properties

Property	Description	Default
file	Location of the file containing the handler chain definition. The location can be relative or absolute with in a	None

Property	Description	Default
	classpath system. If the location is relative, it is relative to the package of the web service. If it is absolute, it is absolute from some path on the classpath.	
`name`	**DEPRECATED** The handler chain name from within the handler chain file.	""

10.2.6.1. Annotation Type Definition

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface HandlerChain {

    String file();

    @Deprecated String name() default "";

}
```

10.2.6.2. Examples

Example 10. `@javax.jws.HandlerChain` - Example

```
@WebService
@HandlerChain(file = "handlers.xml")
public class AddNumbersImpl {
    /**
     * @param number1
     * @param number2
     * @return The sum
     * @throws AddNumbersException if any of the numbers to be added is
     *                             negative.
     */
    public int addNumbers(int number1, int number2) throws
            AddNumbersException {
        if (number1 < 0 || number2 < 0) {
            throw new AddNumbersException("Negative number cant be added " +
                    "!", "Numbers: " + number1 + ", " + number2);
        }
        return number1 + number2;
    }

}
```

Example 11. `@javax.jws.HandlerChain` - Example - handlers.xml

```
<jws:handler-config xmlns:jws="http://java.sun.com/xml/ns/javaee">
    <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>
</jws:handler-config>
```

Important

When using a handler chain file, it is important that the file is store in the appropriate place in the classpath so that the file can be found. This means that when a raw WAR file is created that the file must be place in the proper directory. Please refer to the fromjavahandlers sample application and the Handler for more information.

10.2.7. `@javax.jws.soap.SOAPBinding`

Jakarta Web Services Metadata also allows you to specify a `SOAPBinding` annotation on an endpoint implementation or service endpoint interface. This annotation lets the developer choose between `DOCUMENT\LITERAL WRAPPED`, `DOCUMENT\LITERAL BARE`, `RPC\LITERAL` and `RPC\ENCODED` endpoints with the default being `DOCUMENT\LITERAL WRAPPED`. Jakarta XML Web Services $ws.spec.version; does not support `RPC\ENCODED`. The main difference between `DOCUMENT\LITERAL BARE` and `DOCUMENT\LITERAL WRAPPED` is that methods on a `DOCUMENT\LITERAL WRAPPED` endpoint can have multiple parameters bound to the body of a SOAP message, while a `DOCUMENT\LITERAL BARE` can only have one such parameter. The main difference between `DOCUMENT\LITERAL WRAPPED` and `RPC\LITERAL` is that a `DOCUMENT\LITERAL` invocation can be fully validated by a standard validating XML parser, while an `RPC\LITERAL` invocation cannot because of the implied wrapper element around the invocation body.

Table 8. `@javax.jws.soap.SOAPBinding` - Description of Properties

Property	Description	Default
`style`	Defines the style for messages used in a web service. The value can be either `DOCUMENT` or `RPC`.	`DOCUMENT`
`use`	Defines the encoding used for messages used in web service. Can only be `LITERAL` for Jakarta XML Web Services $ws.spec.version;.	`LITERAL`
`parameterStyle`	Determines if the method's parameters represent the entire message body or whether the parameters are wrapped in a body element named after the operation. Choice of `WRAPPED` or `BARE`. `BARE` can only be used with `DOCUMENT` style bindings.	`WRAPPED`

10.2.7.1. Annotation Type Definition

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface SOAPBinding {

    public enum Style {
        DOCUMENT,
        RPC,
    }

    public enum Use {
        LITERAL,
        ENCODED,
    }

    public enum ParameterStyle {
        BARE,
        WRAPPED,
```

```
    }

    Style style() default Style.DOCUMENT;

    Use use() default Use.LITERAL;

    ParameterStyle parameterStyle() default ParameterStyle.WRAPPED;

}
```

10.2.7.2. Examples

Example 12. `@javax.jws.soap.SOAPBinding` - Example

```
@WebService(targetNamespace = "http://duke.example.org",
        name = "AddNumbers")
@SOAPBinding(style = SOAPBinding.Style.RPC,
        use = SOAPBinding.Use.LITERAL)
public interface AddNumbersIF extends Remote {

    @WebMethod(operationName = "add", action = "urn:addNumbers")
    @WebResult(name = "return")
    public int addNumbers(@WebParam(name = "num1") int number1,
                          @WebParam(name = "num2") int number2) throws
            RemoteException, AddNumbersException;

}
```

10.3. Jakarta XML Web Services Annotations

The following are standard annotations needed by Jakarta XML Web Services that are not defined in Jakarta Web Services Metadata. The developer may not ever use these annotations directly as some of them are generated by Jakarta XML Web Services tools but they will be presented here to avoid confusion.

10.3.1. `@javax.xml.ws.BindingType`

The `BindingType` annotation is used to specify the binding to use for a web service endpoint implementation class. As well as specify additional features that may be enabled.

This annotation may be overriden programmatically or via deployment descriptors, depending on the platform in use.

Table 9. `@javax.xml.ws.BindingType` - Description of Properties

Property	Description	Default

`value`	A binding identifier (a URI).
	
	See the `SOAPBinding` and `HTTPBinding` for the definition of the standard binding identifiers.
	
	`@see javax.xml.ws.Binding`
	
	`@see javax.xml.ws.soap.SOAPBinding#SOAP11HTTP_BINDING`
	
	`@see javax.xml.ws.soap.SOAPBinding#SOAP12HTTP_BINDING`
		`"SOAP 1.1 Protocol"/HTTP`

Property	Description	Default
	@see javax.xml.ws.http.HTTPBinding#HTTP_BINDING	

10.3.1.1. Annotation Type Definition

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface BindingType {
    /**
     * A binding identifier (a URI).
     * If not specified, the default is the SOAP 1.1 / HTTP
     * binding.
     * <p/>
     * See the
     * SOAPBinding and
     * HTTPBinding
     * for the definition of the standard binding identifiers.
     *
     * @see javax.xml.ws.Binding
     * @see javax.xml.ws.soap.SOAPBinding#SOAP11HTTP_BINDING
     * @see javax.xml.ws.soap.SOAPBinding#SOAP12HTTP_BINDING
     * @see javax.xml.ws.http.HTTPBinding#HTTP_BINDING
     */
    String value() default "";

    /**
     * An array of Features to enable/disable on the specified
     * binding.
     * If not specified, features will be enabled/disabled based
     * on their own rules. Refer to the documentation of the
     * feature
     * to determine when it will be automatically enabled.
     * <p/>
     * See the
     * SOAPBinding
     * for the definition of the standard feature identifiers.
     *
     * @see javax.xml.ws.RespectBindingFeature
     * @see javax.xml.ws.soap.AddressingFeature
     * @see javax.xml.ws.soap.MTOMFeature
     * @since JAX-WS 2.1
     */
    Feature[] features() default {};
}
```

10.3.1.2. Examples

Example 13. `@javax.xml.ws.BindingType` - Example

Given the web service defined by

```
@WebService
@BindingType(value = "http://www.w3.org/2003/05/soap/bindings/HTTP/")
public class AddNumbersImpl {
    /**
     * @param number1
```

```
     * @param number2
     * @return The sum
     * @throws AddNumbersException if any of the numbers to be added is
     *                             negative.
     */
    public int addNumbers(int number1, int number2) throws
            AddNumbersException {
        if (number1 < 0 || number2 < 0) {
            throw new AddNumbersException("Negative number cant be " +
                    "added!", "Numbers: " + number1 +
                    ", " + number2);
        }
        return number1 + number2;
    }
}
```

The deployed endpoint would use the SOAP 1.2 over HTTP binding.

10.3.2. `@javax.xml.ws.RequestWrapper`

This annotation annotates methods in the Service Endpoint Interface with the request wrapper bean to be used at runtime.

When starting from Java this annotation is used to resolve overloading conflicts in `DOCUMENT\LITERAL` mode. Only the `className` is required in this case.

Table 10. `@javax.xml.ws.RequestWrapper` - Description of Properties

Property	Description	Default
`localName`	Specifies the `localName` of the XML Schema element representing this request wrapper.	`operationName` as defined by `@javax.jws.WebMethod`
`target-Namespace`	namespace of the request wrapper element.	the `targetNamespace` of the SEI
`className`	The name of the Class representing the request wrapper.	

10.3.2.1. Annotation Type Definition

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface RequestWrapper {
    /**
     * Elements local name.
     */
    public String localName() default "";

    /**
     * Elements namespace name.
     */
    public String targetNamespace() default "";

    /**
     * Request wrapper bean name.
     */
    public String className() default "";
}
```

10.3.2.2. Examples

Example 14. `@javax.xml.ws.RequestWrapper` - Example

```
public interface AddNumbersImpl {
    /**
     * @param arg1
     * @param arg0
     * @return returns int
     * @throws AddNumbersException_Exception
     */
    @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) throws AddNumbersException_Exception;
}
```

10.3.3. `@javax.xml.ws.ResponseWrapper`

This annotation annotates methods in the Service Endpoint Interface with the response wrapper bean to be used at runtime.

When starting from Java this annotation is used to resolve overloading conflicts in `DOCUMENT\LITERAL` mode. Only the `className` is required in this case.

Table 11. `@javax.xml.ws.ResponseWrapper` - Description of Properties

Property	Description	Default
`localName`	Specifies the `localName` of the XML Schema element representing this request wrapper.	`operationName` as defined by `@javax.jws.WebMethod`
`target-Namespace`	namespace of the request wrapper element.	the `targetNamespace` of the SEI
`className`	The name of the Class representing the request wrapper.	

10.3.3.1. Annotation Type Definition

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ResponseWrapper {
    /**
     * Elements local name.
     */
    public String localName() default "";

    /**
     * Elements namespace name.
     */
```

```
    public String targetNamespace() default "";

    /**
     * Request wrapper bean name.
     */
    public String className() default "";
}
```

10.3.3.2. Examples

Example 15. `@javax.xml.ws.ResponseWrapper` - Example

```
public interface AddNumbersImpl {
    /**
     * @param arg1
     * @param arg0
     * @return returns int
     * @throws AddNumbersException_Exception
     */
    @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) throws AddNumbersException_Exception;
}
```

10.3.4. `@javax.xml.ws.ServiceMode`

This annotation allows the Provider developer to indicate whether a `Provider` implementation wishes to work with entire protocol messages or just with protocol message payloads.

Table 12. `@javax.xml.ws.ServiceMode` - Description of Properties

Property	Description	Default
`value`	Convey whether the `Provider` endpoint wants to access then entire message (`MESSAGE`) or just the payload (`PAYLOAD`).	`PAYLOAD`

10.3.4.1. Annotation Type Definition

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface ServiceMode {
    /**
     * Service mode. <code>PAYLOAD</code> indicates that the
     * <code>Provider</code> implementation
     * wishes to work with protocol message payloads only.
     * <code>MESSAGE</code> indicates
     * that the <code>Provider</code> implementation wishes to work with
```

```
     * entire protocol
     * messages.
     */
    public Service.Mode value() default Service.Mode.PAYLOAD;
}
```

10.3.4.2. Examples

Example 16. `@javax.xml.ws.ServiceMode` - Example

```
@ServiceMode(value = Service.Mode.PAYLOAD)
public class AddNumbersImpl implements Provider<Source> {
    public Source invoke(Source source) throws RemoteException {
        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());
            return sendSource(number1, number2);
        } catch (Exception e) {
            e.printStackTrace();
            throw new RemoteException("Error in provider endpoint");
        }
    }

    private Source sendSource(int number1, int number2) {
        int sum = number1 + number2;
        String body = "<ns:addNumbersResponse xmlns:ns =\"http://duke" +
                ".example.org\"><return>" + sum +
                "</return></ns:addNumbersResponse>";
        Source source = new StreamSource(new
 ByteArrayInputStream(body.getBytes()));
        return source;
    }
}
```

10.3.5. `@javax.xml.ws.WebEndpoint`

Used to annotate the `getPortName()` methods of a generated service interface.

The information specified in this annotation is sufficient to uniquely identify a `wsdl:port` element inside a `wsdl:service`. The latter is determined based on the value of the `WebServiceClient` annotation on the generated service interface itself.

Table 13. `@javax.xml.ws.WebEndpoint` - Description of Properties

Property	Description	Default
`name`	Defines the local name of the XML element representing the corresponding port in the WSDL.	""

10.3.5.1. Annotation Type Definition

```
/**
 * Used to annotate the <code>get<em>PortName</em>()</code>
 * methods of a generated service interface.
 * <p/>
 * <p>The information specified in this annotation is sufficient
 * to uniquely identify a <code>wsdl:port</code> element
 * inside a <code>wsdl:service</code>. The latter is
 * determined based on the value of the <code>WebServiceClient</code>
 * annotation on the generated service interface itself.
 *
 * @see javax.xml.ws.WebServiceClient
 * @since JAX-WS 2.0
 */
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface WebEndpoint {
    /**
     * The local name of the endpoint.
     */
    String name() default "";
}
```

10.3.5.2. Examples

Example 17. `@javax.xml.ws.WebEndpoint` - Example

```
@WebServiceClient(name = "AddNumbersImplService",
        targetNamespace = "http://server.fromjava/",
        wsdlLocation = "http://localhost:8080/jaxws-fromjava/addnumbers" +
                "?wsdl")
public class AddNumbersImplService extends Service {
    private final static URL WSDL_LOCATION;
    private final static QName ADDNUMBERSIMPLSERVICE = new QName
            ("http://server.fromjava/", "AddNumbersImplService");
    private final static QName ADDNUMBERSIMPLPORT = new QName
            ("http://server.fromjava/", "AddNumbersImplPort");

    static {
        URL url = null;
        try {
            url = new URL("http://localhost:8080/jaxws-fromjava" +
                    "/addnumbers?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        WSDL_LOCATION = url;
    }

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

    public AddNumbersImplService() {
        super(WSDL_LOCATION, ADDNUMBERSIMPLSERVICE);
    }
```

```
    /**
     * @return returns AddNumbersImpl
     */
    @WebEndpoint(name = "AddNumbersImplPort")
    public AddNumbersImpl getAddNumbersImplPort() {
        return (AddNumbersImpl) super.getPort(ADDNUMBERSIMPLPORT,
 AddNumbersImpl.class);
    }
}
```

10.3.6. `@javax.xml.ws.WebFault`

This annotation is generated by the Jakarta XML Web Services tools into service specific exception classes generated from a WSDL to customize the local and namespace name of the fault element and the name of the fault bean and to mark the service specific exception as one generated from WSDL. The reason that the Jakarta XML Web Services needs to know if a service specific exception is generated from a WSDL or not is because these exceptions will already have a fault bean generated for them. The name of this fault bean is not the same name as the one generated from a Java service specific exception class. For more information on this topic, please refer to section 3.6 of the Jakarta XML Web Services [https://jakarta.ee/specifications/xml-web-services/] specification.

Table 14. `@javax.xml.ws.WebFault` - Description of Properties

Property	Description	Default
`name`	Defines the local name of the XML element representing the corresponding fault in the WSDL.	""
`target-Namespace`	Defines the namespace of the XML element representing the corresponding fault in the WSDL.	""
`faultBean`	The qualified name of the Java class that represents the detail of the fault message.	""

10.3.6.1. Annotation Type Definition

```
/**
 * Used to annotate service specific exception classes to customize
 * to the local and namespace name of the fault element and the name
 * of the fault bean.
 *
 * @since JAX-WS 2.0
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface WebFault {
    /**
     * Element's local name.
     */
    public String name() default "";

    /**
     * Element's namespace name.
     */
    public String targetNamespace() default "";

    /**
```

```
     * Fault bean name.
     */
    public String faultBean() default "";


    /**
     * wsdl:Message's name. Default name is the exception's class name.
     *
     * @since JAX-WS 2.2
     */
    public String messageName() default "";
}
```

10.3.6.2. Examples

Example 18. `@javax.xml.ws.WebFault` - Example

```
@javax.xml.ws.WebFault(name = "AddNumbersException",
        targetNamespace = "http://server.fromjava/jaxws")
public class AddNumbersException_Exception extends Exception {
    private fromjava.client.AddNumbersException faultInfo;

    public AddNumbersException_Exception(String message,
                                         fromjava.client.AddNumbersException
 faultInfo) {
        super(message);
        this.faultInfo = faultInfo;
    }

    public AddNumbersException_Exception(String message,
                                         fromjava.client
                                                 .AddNumbersException
 faultInfo, Throwable cause) {
        super(message, cause);
        this.faultInfo = faultInfo;
    }

    public fromjava.client.AddNumbersException getFaultInfo() {
        return faultInfo;
    }

}
```

10.3.7. `@javax.xml.ws.WebServiceClient`

The information specified in this annotation is sufficient to uniquely identify a `wsdl:service` element inside a WSDL document. This `wsdl:service` element represents the Web service for which the generated service interface provides a client view.

Table 15. `@javax.xml.ws.WebServiceClient` - Description of Properties

Property	Description	Default
`name`	Defines the local name of the `wsdl:serviceName` in the WSDL.	""
`target-Namespace`	Defines the namespace for the `wsdl:serviceName` in the WSDL.	""

Property	Description	Default
wsdlLocation	Specifies the location of the WSDL that defines this service.	""

10.3.7.1. Annotation Type Definition

```
/**
 * Used to annotate a generated service interface.
 * <p/>
 * <p>The information specified in this annotation is sufficient
 * to uniquely identify a <code>wsdl:service</code>
 * element inside a WSDL document. This <code>wsdl:service</code>
 * element represents the Web service for which the generated
 * service interface provides a client view.
 *
 * @since JAX-WS 2.0
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface WebServiceClient {
    /**
     * The local name of the Web service.
     */
    String name() default "";

    /**
     * The namespace for the Web service.
     */
    String targetNamespace() default "";

    /**
     * The location of the WSDL document for the service (a URL).
     */
    String wsdlLocation() default "";
}
```

10.3.7.2. Examples

Example 19. `@javax.xml.ws.WebServiceClient` - Example

```
@WebServiceClient(name = "AddNumbersImplService",
        targetNamespace = "http://server.fromjava/",
        wsdlLocation = "http://localhost:8080/jaxws-fromjava/addnumbers" +
                "?wsdl")
public class AddNumbersImplService extends Service {
    private final static URL WSDL_LOCATION;
    private final static QName ADDNUMBERSIMPLSERVICE = new QName
            ("http://server.fromjava/", "AddNumbersImplService");
    private final static QName ADDNUMBERSIMPLPORT = new QName
            ("http://server.fromjava/", "AddNumbersImplPort");

    static {
        URL url = null;
        try {
            url = new URL("http://localhost:8080/jaxws-fromjava" +
                    "/addnumbers?wsdl");
        } catch (MalformedURLException e) {
```

```
            e.printStackTrace();
        }
        WSDL_LOCATION = url;
    }

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

    public AddNumbersImplService() {
        super(WSDL_LOCATION, ADDNUMBERSIMPLSERVICE);
    }

    /**
     * @return returns AddNumbersImpl
     */
    @WebEndpoint(name = "AddNumbersImplPort")
    public AddNumbersImpl getAddNumbersImplPort() {
        return (AddNumbersImpl) super.getPort(ADDNUMBERSIMPLPORT,
 AddNumbersImpl.class);
    }
}
```

10.3.8. `@javax.xml.ws.WebServiceProvider`

Annotation used to annotate a `Provider` implementation class.

Table 16. `@javax.xml.ws.WebServiceProvider` - Description of Properties

Property	Description	Default
`targetNamespace`	The XML namespace of the the WSDL and some of the XML elements generated from this web service. Most of the XML elements will be in the namespace according to the Jakarta XML Binding mapping rules.	The namespace mapped from the package name containing the web service according to section 3.2 of the Jakarta XML Web Services [https://jakarta.ee/specifications/xml-web-services/] specification.
`serviceName`	The Service name of the web service: `wsdl:service`	The unqualified name of the Java class or interface + "Service"
`portName`	The `wsdl:portName`	
`wsdlLocation`	Location of the WSDL description for the service	

10.3.8.1. Annotation Type Definition

```
/**
 * Used to annotate a Provider implementation class.
 *
 * @since JAX-WS 2.0
 * @see javax.xml.ws.Provider
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
```

```
@Documented
public @interface WebServiceProvider {
    /**
     * Location of the WSDL description for the service.
     */
    String wsdlLocation() default "";

    /**
     * Service name.
     */
    String serviceName() default "";

    /**
     * Target namespace for the service
     */
    String targetNamespace() default "";

    /**
     * Port name.
     */
    String portName() default "";
}
```

10.3.8.2. Examples

Example 20. `@javax.xml.ws.WebServiceProvider` - Example

```
@ServiceMode(value = Service.Mode.PAYLOAD)
@WebServiceProvider(wsdlLocation = "WEB-INF/wsdl/AddNumbers.wsdl")
public class AddNumbersImpl implements Provider {
    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());
            return sendSource(number1, number2);
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException("Error in provider endpoint", e);
        }
    }

    private Source sendSource(int number1, int number2) {
        int sum = number1 + number2;
        String body = "" + sum + "";
        Source source = new StreamSource(new
 ByteArrayInputStream(body.getBytes()));
        return source;
    }
}
```

10.3.9. `@javax.xml.ws.WebServiceRef`

The `WebServiceRef` annotation is used to define a reference to a web service and (optionally) an injection target for it. Web service references are resources in the Jakarta EE sense.

Table 17. `@javax.xml.ws.WebServiceRef` - Description of Properties

Property	Description	Default
`name`	The JNDI name of the resource. For field annotations, the default is the field name. For method annotations, the default is the JavaBeans property name corresponding to the method. For class annotations, there is no default and this must be specified.	
`type`	The Java type of the resource. For field annotations, the default is the type of the field. For method annotations, the default is the type of the JavaBeans property. For class annotations, there is no default and this must be specified.	
`mappedName`	A product specific name that this resource should be mapped to.	
`value`	The service class, always a type extending `javax.xml.ws.Service`. This element must be specified whenever the type of the reference is a service endpoint interface.	
`wsdlLocation`	Location of the WSDL description for the service	

10.3.9.1. Annotation Type Definition

```
/**
 * The <code>WebServiceRef</code> annotation is used to
 * define a reference to a web service and
 * (optionally) an injection target for it.
 * It can be used to inject both service and proxy
 * instances. These injected references are not thread safe.
 * If the references are accessed by multiple threads,
 * usual synchronization techinques can be used to
 * support multiple threads.
 * <p/>
 * Web service references are resources in the Jakarta EE sense.
 * The annotations (for example, {@link Addressing}) annotated with
 * meta-annotation {@link WebServiceFeatureAnnotation}
 * can be used in conjunction with <code>WebServiceRef</code>.
 * The created reference MUST be configured with annotation's web service
 * feature.
 * <p/>
 * If a Jakarta XML Web Services implementation encounters an unsupported or
unrecognized
 * annotation annotated with the <code>WebServiceFeatureAnnotation</code>
 * that is specified with <code>WebServiceRef</code>,
 * an ERROR MUST be given.
 *
 * @see javax.annotation.Resource
 * @see WebServiceFeatureAnnotation
```

```
 * @since JAX-WS 2.0
 */
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface WebServiceRef {
    /**
     * The JNDI name of the resource.  For field annotations,
     * the default is the field name.  For method annotations,
     * the default is the JavaBeans property name corresponding
     * to the method.  For class annotations, there is no default
     * and this MUST be specified.
     * <p/>
     * The JNDI name can be absolute(with any logical namespace) or
     * relative
     * to JNDI <code>java:comp/env</code> namespace.
     */
    String name() default "";

    /**
     * The Java type of the resource.  For field annotations,
     * the default is the type of the field.  For method annotations,
     * the default is the type of the JavaBeans property.
     * For class annotations, there is no default and this MUST be
     * specified.
     */
    Class<?> type() default Object.class;

    /**
     * A product specific name that this resource should be mapped to.
     * The name of this resource, as defined by the <code>name</code>
     * element or defaulted, is a name that is local to the application
     * component using the resource.  (When a relative JNDI name
     * is specified, then it's a name in the JNDI
     * <code>java:comp/env</code> namespace.)  Many application servers
     * provide a way to map these local names to names of resources
     * known to the application server.  This mapped name is often a
     * <i>global</i> JNDI name, but may be a name of any form.
     * <p/>
     * Application servers are not required to support any particular
     * form or type of mapped name, nor the ability to use mapped names.
     * The mapped name is product-dependent and often
     * installation-dependent.
     * No use of a mapped name is portable.
     */
    String mappedName() default "";

    /**
     * The service class, alwiays a type extending
     * <code>javax.xml.ws.Service</code>. This element MUST be specified
     * whenever the type of the reference is a service endpoint interface.
     */
    // 2.1 has Class value() default Object.class;
    // Fixing this raw Class type correctly in 2.2 API. This shouldn't
    // cause
    // any compatibility issues for applications.
    Class<? extends Service> value() default Service.class;

    /**
     * A URL pointing to the WSDL document for the web service.
```

```
     * If not specified, the WSDL location specified by annotations
     * on the resource type is used instead.
     */
    String wsdlLocation() default "";

    /**
     * A portable JNDI lookup name that resolves to the target
     * web service reference.
     *
     * @since JAX-WS 2.2
     */
    String lookup() default "";

}
```

10.3.10. `@javax.xml.ws.Action`

The `Action` annotation allows explicit association of `Action` message addressing property with `input`, `output`, and `fault` messages of the mapped WSDL operation.

This annotation can be specified on each method of a service endpoint interface or implementation. For such a method, the mapped operation in the generated WSDL contains explicit `wsaw:Action` attribute on the WSDL `input`, `output` and `fault` messages of the WSDL `operation` based upon which attributes of the `Action` annotation have been specified.

Table 18. `@javax.xml.ws.Action` - Description of Properties

Property	Description	Default
`input`	Explicit value of `Action` message addressing property for the `input` message of the operation. If the value is "", then no `wsaw:Action` is generated.	""
`ouput`	Explicit value of `Action` message addressing property for the `output` message of the operation. If the value is "", then no `wsaw:Action` is generated.	""
`fault`	Explicit value of `Action` message addressing property for the `fault` message(s) of the operation. Each exception that is mapped to a fault and requires explicit `Action` message addressing property, need to be specified as a value in this property using FaultAction annotation.	{}

10.3.10.1. Annotation Type Definition

```
/**
 * The <code>Action</code> annotation allows explicit association of a
 * WS-Addressing <code>Action</code> message addressing property with
 * <code>input</code>, <code>output</code>, and
 * <code>fault</code> messages of the mapped WSDL operation.
 * <p/>
 * This annotation can be specified on each method of a service endpoint
 * interface.
 * For such a method, the mapped operation in the generated WSDL's
 * <code>wsam:Action</code> attribute on the WSDL <code>input</code>,
 * <code>output</code> and <code>fault</code> messages of the WSDL
 * <code>operation</code>
 * is based upon which attributes of the <code>Action</code> annotation
```

```
 * have been specified.
 * For the exact computation of <code>wsam:Action</code> values for the
 * messages, refer
 * to the algorithm in the Jakarta XML Web Services specification.
 *
 * @see FaultAction
 * @since JAX-WS 2.1
 */

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Action {
    /**
     * Explicit value of the WS-Addressing <code>Action</code> message
     * addressing property for the <code>input</code>
     * message of the operation.
     */
    String input() default "";

    /**
     * Explicit value of the WS-Addressing <code>Action</code> message
     * addressing property for the <code>output</code>
     * message of the operation.
     */
    String output() default "";

    /**
     * Explicit value of the WS-Addressing <code>Action</code> message
     * addressing property for the <code>fault</code>
     * message(s) of the operation. Each exception that is mapped to a
     * fault and requires an explicit WS-Addressing
     * <code>Action</code> message addressing property,
     * needs to be specified as a value in this property
     * using {@link FaultAction} annotation.
     */
    FaultAction[] fault() default {};
}
```

10.3.10.2. Examples

Example 21. `@javax.xml.ws.Action` - Example 1 - Specify explicit values for `Action` message addressing property for `input` and `output` messages.

```
@javax.jws.WebService
public class AddNumbersImpl {
    @javax.xml.ws.Action(
            input = "http://example.com/inputAction",
            output = "http://example.com/outputAction")
    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

The generated WSDL looks like:

```
<definitions targetNamespace="http://example.com/numbers" ...>

    ...
```

```
    <portType name="AddNumbersPortType">
        <operation name="AddNumbers">
            <input message="tns:AddNumbersInput" name="Parameters"
                    wsaw:Action="http://example.com/inputAction"/>
            <output message="tns:AddNumbersOutput" name="Result"
                    wsaw:Action="http://example.com/outputAction"/>
        </operation>
    </portType>

    ...

</definitions>
```

Example 22. `@javax.xml.ws.Action` - Example 2 - Specify explicit value for `Action` message addressing property for only the `input` message.

The default values are used for the `output` message.

```
@javax.jws.WebService
public class AddNumbersImpl {
    @javax.xml.ws.Action(input = "http://example.com/inputAction")
    public int addNumbers(int number1, int number2) {
        return number1 + number2;
    }
}
```

The generated WSDL looks like:

```
<definitions targetNamespace="http://example.com/numbers" ...>

    ...

    <portType name="AddNumbersPortType">
        <operation name="AddNumbers">
            <input message="tns:AddNumbersInput" name="Parameters"
                    wsaw:Action="http://example.com/inputAction"/>
            <output message="tns:AddNumbersOutput" name="Result"/>
        </operation>
    </portType>

    ...

</definitions>
```

It is legitimate to specify an explicit value for `Action` message addressing property for `output` message only. In this case, a default value of `wsaw:Action` is used for the `input` message.

Example 23. `@javax.xml.ws.Action` - Example 3 - `@FaultAction`

See `@javax.xml.ws.FaultAction` for an example of how to specify an explicit value for `Action` message addressing property for the `fault` message.

10.3.11. `@javax.xml.ws.FaultAction`

The `FaultAction` annotation is used inside an `Action` annotation to allow an explicit association of `Action` message addressing property with the `fault` messages of the WSDL operation mapped from the exception class.

The `fault` message in the generated WSDL operation mapped for `className` class contains explicit `wsaw:Action` attribute.

Table 19. `@javax.xml.ws.FaultAction` - Description of Properties

Property	Description	Default
`className`	Name of the exception class	there is no default and is required.
`value`	Value of `Action` message addressing property for the exception	""

10.3.11.1. Annotation Type Definition

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface FaultAction {
    /**
     * Name of the exception class
     */
    Class<? extends Exception> className();

    /**
     * Value of WS-Addressing <code>Action</code> message addressing
     * property for the exception
     */
    String value() default "";
}
```

10.3.11.2. Examples

Example 24. `@javax.xml.ws.FaultAction` - Example 1 - Specify explicit values for `Action` message addressing property for the `input`, `output` and `fault` message if the Java method throws only one service specific exception.

```
@javax.jws.WebService
public class AddNumbersImpl {
    @javax.xml.ws.Action(
            input = "http://example.com/inputAction",
            output = "http://example.com/outputAction",
            fault = {
                    @javax.xml.ws.FaultAction(className =
                            AddNumbersException.class,
                            value = "http://example.com/faultAction")})
    public int addNumbers(int number1, int number2) throws
            AddNumbersException {
        return number1 + number2;
    }
}
```

The generated WSDL looks like:

```
<definitions targetNamespace="http://example.com/numbers" ...>

    ...

    <portType name="AddNumbersPortType">
```

```
        <operation name="AddNumbers">
            <input message="tns:AddNumbersInput" name="Parameters"
                    wsaw:Action="http://example.com/inputAction"/>
            <output message="tns:AddNumbersOutput" name="Result"
                    wsaw:Action="http://example.com/outputAction"/>
            <fault message="tns:AddNumbersException"
                    name="AddNumbersException"
                    wsaw:Action="http://example.com/faultAction"/>
        </operation>
    </portType>

    ...

</definitions>
```

Example 25. `@javax.xml.ws.FaultAction` - Example 1 - Specify explicit values for `Action` message addressing property if the Java method throws only one service specific exception, without specifying the values for `input` and `output` messages.

```
@javax.jws.WebService
public class AddNumbersImpl {
    @javax.xml.ws.Action(
            fault = {@javax.xml.ws.FaultAction(className =
                    AddNumbersException.class,
                    value = "http://example.com/faultAction")})
    public int addNumbers(int number1, int number2) throws
            AddNumbersException {
        return number1 + number2;
    }
}
```

The generated WSDL looks like:

```
<definitions targetNamespace="http://example.com/numbers" ...>

    ...

    <portType name="AddNumbersPortType">
        <operation name="AddNumbers">
            <input message="tns:AddNumbersInput" name="Parameters"/>
            <output message="tns:AddNumbersOutput" name="Result"/>
            <fault message="tns:addNumbersFault" name="InvalidNumbers"
                    wsa:Action="http://example.com/addnumbers/fault"/>
        </operation>
    </portType>

    ...

</definitions>
```

Example 26. `@javax.xml.ws.FaultAction` - Example 1 - Specify explicit values for `Action` message addressing property if the Java method throws more than one service specific exception.

```
@javax.jws.WebService
public class AddNumbersImpl {
    @javax.xml.ws.Action(
```

```
            fault = {@javax.xml.ws.FaultAction(className =
                    AddNumbersException.class,
                    value = "http://example.com/addFaultAction"),

                    @javax.xml.ws.FaultAction(className =
                            TooBigNumbersException.class,
                            value = "http://example" +
                                    ".com/toobigFaultAction")})
    public int addNumbers(int number1, int number2) throws
 AddNumbersException, TooBigNumbersException {
        return number1 + number2;
    }
}
```

The generated WSDL looks like:

```
<definitions targetNamespace="http://example.com/numbers" ...>

    ...

    <portType name="AddNumbersPortType">
        <operation name="AddNumbers">
            <input message="tns:AddNumbersInput" name="Parameters"/>
            <output message="tns:AddNumbersOutput" name="Result"/>
            <fault message="tns:addNumbersFault" name="AddNumbersException"
                    wsa:Action="http://example.com/addnumbers/fault"/>
            <fault message="tns:tooBigNumbersFault"
                    name="TooBigNumbersException"
                    wsa:Action="http://example.com/toobigFaultAction"/>
        </operation>
    </portType>

    ...

</definitions>
```

10.4. Jakarta XML Binding Annotations

The following Jakarta XML Binding annotations are being documented because Eclipse Implementation of XML Web Services generates them when generating wrapper beans and exception beans according to the Jakarta XML Web Services spec. Please refer to sections 3.5.2.1 and 3.6 of the Jakarta XML Web Services [https://jakarta.ee/specifications/xml-web-services/] specification for more information on these beans. For more information on these and other Jakarta XML Binding annotations please refer to the Jakarta XML Binding specification [https://jakarta.ee/specifications/xml-binding/].

10.4.1. `@javax.xml.bind.annotation.XmlRootElement`

This annotation is used to map a top level class to a global element in the XML schema used by the WSDL of the web service.

Table 20. `@javax.xml.bind.annotation.XmlRootElement` - Description of Properties

Property	Description	Default
`name`	Defines the local name of the XML element representing the annotated class	`##default` – the name is derived from the class

Property	Description	Default
`namespace`	Defines the namespace of the XML element representing the annotated class	`##default` – the namespace is derived from the package of the class

10.4.1.1. Annotation Type Definition

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface XmlRootElement {
    /**
     * namespace name of the XML element.
     * <p/>
     * If the value is "##default", then the XML namespace name is
     * derived
     * from the package of the class ( {@link XmlSchema} ). If the
     * package is unnamed, then the XML namespace is the default
     * empty
     * namespace.
     */
    String namespace() default "##default";

    /**
     * local name of the XML element.
     * <p/>
     * If the value is "##default", then the name is derived from
     * the
     * class name.
     */
    String name() default "##default";

}
```

10.4.1.2. Examples

Example 27. `@javax.xml.bind.annotation.XmlRootElement` - Example

```
@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;
    }
}
```

10.4.2. `@javax.xml.bind.annotation.XmlAccessorType`

This annotation is used to specify whether fields or properties are serialized by default.

Table 21. `@javax.xml.bind.annotation.XmlAccessorType` - Description of Properties

Property	Description	Default
`value`	Specifies whether fields or properties are serialized by default. The value can be `XmlAccessType.FIELD` or `XmlAccessType.PROPERTY` or `XmlAccessType.PUBLIC_MEMBER` or `XmlAccessType.NONE`	`XmlAccessType.PROPERTY`

10.4.2.1. Annotation Type Definition

```
@Inherited
@Retention(RUNTIME)
@Target({PACKAGE, TYPE})
public @interface XmlAccessorType {

    /**
     * Specifies whether fields or properties are serialized.
     *
     * @see XmlAccessType
     */
    XmlAccessType value() default XmlAccessType.PUBLIC_MEMBER;
}

/**
 * Used by XmlAccessorType to control serialization of fields or
 * properties.
 */
public enum XmlAccessType {
    /**
     * Every getter/setter pair in a Jakarta XML Binding-bound class will be
     * automatically
     * bound to XML, unless annotated by {@link XmlTransient}.
     * <p/>
     * Fields are bound to XML only when they are explicitly
     * annotated
     * by some of the Jakarta XML Binding annotations.
     */
    PROPERTY,
    /**
     * Every non static, non transient field in a Jakarta XML Binding-bound
 class
     * will be automatically
     * bound to XML, unless annotated by {@link XmlTransient}.
```

```
     * <p/>
     * Getter/setter pairs are bound to XML only when they are
     * explicitly annotated
     * by some of the Jakarta XML Binding annotations.
     */
    FIELD,
    /**
     * Every public getter/setter pair and every public field will
     * be
     * automatically bound to XML, unless annotated by {@link
     * XmlTransient}.
     * <p/>
     * Fields or getter/setter pairs that are private, protected,
     * or
     * defaulted to package-only access are bound to XML only when
     * they are
     * explicitly annotated by the appropriate Jakarta XML Binding
 annotations.
     */
    PUBLIC_MEMBER,
    /**
     * None of the fields or properties is bound to XML unless they
     * are specifically  annotated with some of the Jakarta XML Binding
     * annotations.
     */
    NONE
}
```

10.4.2.2. Examples

Example 28. `@javax.xml.bind.annotation.XmlAccessorType` - Example

```
@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;
    }
```

```
}
```

10.4.3. `@javax.xml.bind.annotation.XmlType`

This annotation is used to map a value class to an XML Schema type. A value class is a data container for values represented by properties and fields. A schema type is a data container for values represented by schema components within a schema type's content model (e.g. Model groups, attributes etc).

Table 22. `@javax.xml.bind.annotation.XmlType` - Description of Properties

Property	Description	Default
`name`	Defines the local name of the XML type representing this class in the XML schema used by the WSDL of the web service	"`##default`"
`namespace`	Defines the namespace of the XML type representing this class in the XML schema used by the WSDL of the web service	"`##default`"
`propOrder`	Defines a list of names of JavaBean properties in the class. Each name in the list is the name of a Java identifier of the JavaBean property. The order in which JavaBean properties are listed is the order of XML Schema elements to which the JavaBean properties are mapped. All of the JavaBean properties being mapped must be listed (i.e. if a JavaBean property mapping is prevented by `@XmlTransient` then it does not have to be listed). Otherwise, it is an error. By default, the JavaBean properties are ordered using a default order specified in the Jakarta XML Binding 2.3 specification [https://jakarta.ee/specifications/xml-binding/].	{""}

10.4.3.1. Annotation Type Definition

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface XmlType {
    /**
     * Name of the XML Schema type which the class is mapped.
     */
    String name() default "##default";

    /**
     * Specifies the order for XML Schema elements when class is
     * mapped to a XML Schema complex type.
     * <p/>
     * <p> Refer to the table for how the propOrder affects the
     * mapping of class </p>
     * <p/>
     * <p> The propOrder is a list of names of JavaBean properties in
     * the class. Each name in the list is the name of a Java
     * identifier of the JavaBean property. The order in which
     * JavaBean properties are listed is the order of XML Schema
     * elements to which the JavaBean properties are mapped. </p>
     * <p> All of the JavaBean properties being mapped to XML Schema
```

```
     * elements
     * must be listed.
     * <p> A JavaBean property or field listed in propOrder must not
     * be transient or annotated with <tt>@XmlTransient</tt>.
     * <p> The default ordering of JavaBean properties is determined
     * by @{@link XmlAccessorOrder}.
     */
    String[] propOrder() default {""};

    /**
     * Name of the target namespace of the XML Schema type. By
     * default, this is the target namespace to which the package
     * containing the class is mapped.
     */
    String namespace() default "##default";

    /**
     * Class containing a no-arg factory method for creating an
     * instance of this class. The default is this class.
     * <p/>
     * <p>If <tt>factoryClass</tt> is DEFAULT.class and
     * <tt>factoryMethod</tt> is "", then there is no static factory
     * method.
     * <p/>
     * <p>If <tt>factoryClass</tt> is DEFAULT.class and
     * <tt>factoryMethod</tt> is not "", then
     * <tt>factoryMethod</tt> is the name of a static factory method
     * in this class.
     * <p/>
     * <p>If <tt>factoryClass</tt> is not DEFAULT.class, then
     * <tt>factoryMethod</tt> must not be "" and must be the name of
     * a static factory method specified in <tt>factoryClass</tt>.
     */
    Class factoryClass() default DEFAULT.class;

    /**
     * Used in {@link XmlType#factoryClass()} to
     * signal that either factory mehod is not used or
     * that it's in the class with this {@link XmlType} itself.
     */
    static final class DEFAULT {
    }

    /**
     * Name of a no-arg factory method in the class specified in
     * <tt>factoryClass</tt> factoryClass().
     */
    String factoryMethod() default "";
}
```

10.4.3.2. Examples

Example 29. `@javax.xml.bind.annotation.XmlType` - Example

```
@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;
    }
}
```

10.4.4. `@javax.xml.bind.annotation.XmlElement`

This annotation is used to map a property contained in a class to a local element in the XML Schema complex type to which the containing class is mapped.

Table 23. `@javax.xml.bind.annotation.XmlElement` - Description of Properties

Property	Description	Default
`name`	Defines the local name of the XML element representing the property of a JavaBean	"`##default`" - the element name is derived from the JavaBean property name.
`namespace`	Defines the namespace of the XML element representing the property of a JavaBean	"`##default`" - the namespace of the containing class
`nillable`	Not generated by Jakarta XML Web Services	
`type`	Not generated by Jakarta XML Web Services	

10.4.4.1. Annotation Type Definition

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})
public @interface XmlElement {
    /**
     * Name of the XML Schema element.
     * <p> If the value is "##default", then element name is derived from
     * the
     * JavaBean property name.
     */
    String name() default "##default";

    /**
     * Customize the element declaration to be nillable.
     * <p>If nillable() is true, then the JavaBean property is
```

```
     * mapped to a XML Schema nillable element declaration.
     */
    boolean nillable() default false;

    /**
     * Customize the element declaration to be required.
     * <p>If required() is true, then Javabean property is mapped to
     * an XML schema element declaration with minOccurs="1".
     * maxOccurs is "1" for a single valued property and "unbounded"
     * for a multivalued property.
     * <p>If required() is false, then the Javabean property is mapped
     * to XML Schema element declaration with minOccurs="0".
     * maxOccurs is "1" for a single valued property and "unbounded"
     * for a multivalued property.
     */

    boolean required() default false;

    /**
     * XML target namespace of the XML Schema element.
     * <p/>
     * If the value is "##default", then the namespace is determined
     * as follows:
     * <ol>
     * <li>
     * If the enclosing package has {@link XmlSchema} annotation,
     * and its {@link XmlSchema#elementFormDefault() elementFormDefault}
     * is {@link XmlNsForm#QUALIFIED QUALIFIED}, then the namespace of
     * the enclosing class.
     * <p/>
     * <li>
     * Otherwise &#39;&#39; (which produces unqualified element in the
     * default
     * namespace.
     * </ol>
     */
    String namespace() default "##default";

    /**
     * Default value of this element.
     * <p/>
     * <p/>
     * The <pre>'\u0000'</pre> value specified as a default of this
     * annotation element
     * is used as a poor-man's substitute for null to allow implementations
     * to recognize the 'no default value' state.
     */
    String defaultValue() default "\u0000";

    /**
     * The Java class being referenced.
     */
    Class type() default DEFAULT.class;

    /**
     * Used in {@link XmlElement#type()} to
     * signal that the type be inferred from the signature
     * of the property.
     */
    static final class DEFAULT {
```

```
    }
}
```

10.4.4.2. Examples

Example 30. `@javax.xml.bind.annotation.XmlElement` - Example

```
@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;
    }
}
```

10.4.5. `@javax.xml.bind.annotation.XmlSeeAlso`

Instructs Jakarta XML Binding to also bind other classes when binding this class.

Table 24. `@javax.xml.bind.annotation.XmlSeeAlso` - Description of Properties

Property	Description	Default
`value`	Other classes that Jakarta XML Binding can use when binding this class	{}

10.4.5.1. Annotation Type Definition

```
/**
 * Instructs Jakarta XML Binding to also bind other classes when binding this
 class.
 * <p/>
 * Java makes it impractical/impossible to list all sub-classes of
 * a given class. This often gets in a way of Jakarta XML Binding users, as
 it Jakarta XML Binding
 * cannot automatically list up the classes that need to be known
 * to {@link JAXBContext}.
```

```
 * <p/>
 * For example, with the following class definitions:
 * <p/>
 * <pre>
 * class Animal {}
 * class Dog extends Animal {}
 * class Cat extends Animal {}
 * </pre>
 * <p/>
 * The user would be required to create {@link JAXBContext} as
 * <tt>JAXBContext.newInstance(Dog.class,Cat.class)</tt>
 * (<tt>Animal</tt> will be automatically picked up since <tt>Dog</tt>
 * and <tt>Cat</tt> refers to it.)
 * <p/>
 * {@link XmlSeeAlso} annotation would allow you to write:
 * <pre>
 * &#64;XmlSeeAlso({Dog.class,Cat.class})
 * class Animal {}
 * class Dog extends Animal {}
 * class Cat extends Animal {}
 * </pre>
 * <p/>
 * This would allow you to do <tt>JAXBContext.newInstance(Animal.class)
 * </tt>.
 * By the help of this annotation, Jakarta XML Binding implementations will
 be able to
 * correctly bind <tt>Dog</tt> and <tt>Cat</tt>.
 *
 * @author Kohsuke Kawaguchi
 * @since JAXB2.1
 */
@Target({ElementType.TYPE})
@Retention(RUNTIME)
public @interface XmlSeeAlso {
    Class[] value();
}
```

10.5. JSR 250 (Common Annotations) Annotations

The following annotations are being documented because Jakarta XML Web Services endpoints use them for resource injection, and as lifecycle methods. Please refer to sections 5.2.1 and 5.3 of the Jakarta XML Web Services [https://jakarta.ee/specifications/xml-web-services/] specification for resource injection, and lifecycle management. For more information on these and other common annotations please refer to the *Jakarta Annotations* [https://jakarta.ee/specifications/annotations/].

10.5.1. `@javax.annotation.Resource`

This annotation is used to mark a WebServiceContext resource that is needed by a web service. It is applied to a field or a method for Jakarta XML Web Services endpoints. The container will inject an instance of the WebServiceContext resource into the endpoint implementation when it is initialized.

Table 25. `@javax.annotation.Resource` - Description of Properties

Property	Description	Default
`type`	Java type of the resource	For field annotations, the default is the type of the field. For method annotations, the

Property	Description	Default
		default is the type of the JavaBeans property.

10.5.1.1. Annotation Type Definition

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Resource {

    // ...

    /**
     * The Java type of the resource.  For field annotations,
     * the default is the type of the field.  For method annotations,
     * the default is the type of the JavaBeans property.
     * For class annotations, there is no default and this must be
     * specified.
     */
    Class type() default java.lang.Object.class;
}
```

10.5.1.2. Examples

Example 31. `@javax.annotation.Resource` - Example

```
@WebService
public class HelloImpl {
    @Resource
    private WebServiceContext context;

    public String echoHello(String name) {
        // ...
    }
}
```

10.5.2. `@javax.annotation.PostConstruct`

This annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization. This method MUST be invoked before the class is put into service.

10.5.2.1. Annotation Type Definition

```
/**
 * The PostConstruct annotation is used on a method that needs to be
 * executed
 * after dependency injection is done to perform any initialization. This
 * method MUST be invoked before the class is put into service. This
 * annotation MUST be supported on all classes that support dependency
 * injection. The method annotated with PostConstruct MUST be invoked even
 * if the class does not request any resources to be injected. Only one
 * method can be annotated with this annotation. The method on which the
 * PostConstruct annotation is applied MUST fulfill all of the following
 * criteria -
 * - The method MUST NOT have any parameters except in the case of EJB
 * interceptors   in which case it takes an InvocationC ontext object as
 * defined by the EJB   specification.
 * - The return type of the method MUST be void.
```

```
 * - The method MUST NOT throw a checked exception.
 * - The method on which PostConstruct is applied MAY be public, protected,
 * package private or private.
 * - The method MUST NOT be static except for the application client.
 * - The method MAY be final.
 * - If the method throws an unchecked exception the class MUST NOT be
 * put into
 * service except in the case of EJBs where the EJB can handle exceptions
 * and
 * even   recover from them.
 *
 * @see javax.annotation.PreDestroy
 * @see javax.annotation.Resource
 * @since Common Annotations 1.0
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface PostConstruct {
}
```

10.5.2.2. Examples

Example 32. `@javax.annotation.PostConstruct` - Example

```
@WebService
public class HelloImpl {
    @PostConstruct
    private void init() {
        // ...
    }

    public String echoHello(String name) {
        // ...
    }
}
```

10.5.3. `@javax.annotation.PreDestroy`

The PreDestroy annotation is used on methods as a callback notification to signal that the instance is in the process of being removed by the container. The method annotated with PreDestroy is typically used to release resources that it has been holding.

10.5.3.1. Annotation Type Definition

```
/**
 * The PreDestroy annotation is used on methods as a callback
 * notification to
 * signal that the instance is in the process of being removed by the
 * container. The method annotated with PreDestroy is typically used to
 * release resources that it has been holding. This annotation MUST be
 * supported by all container managed objects that support PostConstruct
 * except the application client container in Jakarta EE. The method on
 * which
 * the PreDestroy annotation is applied MUST fulfill all of the following
 * criteria -
 * - The method MUST NOT have any parameters except in the case of EJB
 * interceptors in which case it takes an InvocationContext object as
```

```
 * defined
 * by the EJB specification.
 * - The return type of the method MUST be void.
 * - The method MUST NOT throw a checked exception.
 * - The method on which PreDestroy is applied MAY be public, protected,
 * package private or private.
 * - The method MUST NOT be static.
 * - The method MAY be final.
 * - If the method throws an unchecked exception it is ignored except in
 * the
 * case of EJBs where the EJB can handle exceptions.
 *
 * @see javax.annotation.PostConstruct
 * @see javax.annotation.Resource
 * @since Common Annotations 1.0
 */

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface PreDestroy {
}
```

10.5.3.2. Examples

Example 33. `@javax.annotation.PreDestroy` - Example

```
@WebService
public class HelloImpl {
    public String echoHello(String name) {
        // ...
    }

    @PreDestroy
    private void release() {
        // ...
    }
}
```

11. WS-Addressing

11.1. WS-Addressing in Eclipse Implementation of XML Web Services

Web Services Addressing [http://www.w3.org/2002/ws/addr/] provides transport-neutral mechanisms to address Web services and messages. Jakarta XML Web Services specification requires support for W3C Core [http://www.w3.org/TR/ws-addr-core], SOAP Binding [http://www.w3.org/TR/ws-addr-soap] and Addressing 1.0 - Metadata [http://www.w3.org/TR/2007/REC-ws-addr-metadata-20070904] specifications and defines standard API to enable/disable W3C WS-Addressing on the client and service endpoint. In addition to that, Eclipse Implementation of XML Web Services also supports Member Submission [http://www.w3.org/Submission/ws-addressing/] version of WS-Addressing. The member submission version is supported in an implementation specific way. For compatility with Jakarta XML Web Services 2.1 behavior, Eclipse Implementation of XML Web Services 2.3.3 also supports wsdls conforming to WSDL Binding [http://www.w3.org/TR/ws-addr-wsdl] specification. The subsequent sections describe how the two WS-Addressing versions can be enabled/disabled on client and server side .

11.2. Why WS-Addressing?

The subsequent sections explain the different use cases served by WS-Addressing.

11.2.1. Transport Neutrality

This section describes how a message can be sent to a Web service endpoint in transport neutral manner.

Example 34. SOAP 1.2 message, without WS-Addressing, sent over HTTP

```
POST /fabrikam/Purchasing HTTP 1.1/POST                    ❶
Host: example.com
SOAPAction: http://example.com/fabrikam/SubmitPO

<S:Envelope                                                ❷
        xmlns:S="http://www.w3.org/2003/05/soap-envelope"
        xmlns:wombat="http://wombat.org/">
    <S:Header>
        <wombat:MessageID>
            uuid:e197db59-0982-4c9c-9702-4234d204f7f4
        </wombat:MessageID>
    </S:Header>
    <S:Body>
        ...
    </S:Body>
</S:Envelope>
```

❶ HTTP transport headers.
❷ SOAP message in HTTP body.

The host (`example.com`), the dispatch method (`POST`) and the URL to dispatch to (`/fabrikam/Purchasing`) are in the HTTP transport headers. The actual message and implied meaning (for example payload's `QName` or `SOAPAction`) is defined by the messaging system (`SOAP`) or transport protocol (`HTTP`). If the message is to be sent over an alternate transport, such as SMTP, then the information conveyed in HTTP transport headers need to be mapped to SMTP specific headers. On the server side, to dispatch successfully, a Web service stack has to gather the information from the SMTP (as opposed to HTTP) headers and the SOAP message.

Also in the above message, there is no standard header to establish the identity of a message. In this case, `MessageID` header defined in the namespace URI bound to `wombat` prefix is used but is application specific and is thus not re-usable.

WS-Addressing introduce Message Addressing Properties that collectively augment a message to normalize this information.

Example 35. SOAP 1.2 message, with WS-Addressing, sent over HTTP

```
POST /fabrikam/Purchasing HTTP 1.1/POST                    ❶
Host: example.com
SOAPAction: http://example.com/fabrikam/SubmitPO

<S:Envelope                                                ❷
        xmlns:S="http://www.w3.org/2003/05/soap-envelope"
        xmlns:wsa="http://www.w3.org/2005/08/addressing/">
    <S:Header>
        <wsa:MessageID>                                    ❸
```

```
            uuid:e197db59-0982-4c9c-9702-4234d204f7f4
        </wsa:MessageID>
        <wsa:To>
            http://example.com/fabrikam/Purchasing
        </wsa:To>
        <wsa:Action>
            http://example.com/fabrikam/SubmitPO
        </wsa:Action>
    </S:Header>
    <S:Body>
        ...
    </S:Body>
</S:Envelope>
```

❶ HTTP transport headers.
❷ SOAP message in HTTP body.
❸ Binding of Message Addressing Properties to SOAP 1.2 message.

For example, `wsa:MessageID` is a binding of an abstract property that defines an absolute URI that uniquely identifies the message, `wsa:To` is binding of an abstract absolute URI representing the address of the intended receiver of this message and `wsa:Action` is binding of an abstract absolute IRI that uniquely identifies the semantics implied by this message. All the information earlier shared between transport protocols and messaging systems is now normalized into a uniform format that can be processed independent of transport or application.

If the exactly same message is to be sent/received using a different transport, for example asynchronously over SMTP, then the value of `wsa:To` header could be changed to `mailto:purchasing@example.com`. The updated `wsa:To` header looks like:

```
<wsa:To>
    mailto:purchasing@example.com
</wsa:To>
```

On the server side, Web services stack can gather all the information from the SOAP message and then dispatch it correctly.

11.2.2. Stateful Web service

Web services are usually stateless, i.e. the service endpoint receives a request and responds back without saving any processing state in between different requests. However making Web services stateful enables to share multiple instances of service endpoints. For example, consider a stateful Bank Web service. The client (say bank customer) can obtain a bank EPR, with relevant state information stored as reference parameters, and invoke a method on that EPR to do a series of banking operations. On the service endpoint, whenever a request is received, the reference parameters from the EPR are available as first-class SOAP headers allowing the endpoint to restore the state.

Eclipse Implementation of XML Web Services 2.3.3 enables stateful Web services to be annotated with `com.sun.xml.ws.developer.Stateful` annotation.

11.2.3. Simple and Complex MEPs

WS-Addressing defines standard Message Addressing Properties [http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/#abstractmaps] (MAPs) to support simple and complex message patterns. The SOAP Binding defines a mapping of these MAPs to SOAP headers and convey end-to-end message characteristics including addressing for source and destination endpoints as well as message identity. For example `destination` MAP represents an absolute IRI representing the address of the intended receiver of the message and is mapped to a SOAP header with `wsa:To` element name. `reply endpoint` represents

an endpoint reference for the intended receiver for replies to this message and is mapped to a SOAP header with `wsa:ReplyTo` element name. In addition, WSDL Binding, also defines requirement on the presence of these MAPs for standard Message Exchange Patterns (MEPs) such as request/response [http://www.w3.org/TR/wsdl#_request-response] and one-way [http://www.w3.org/TR/wsdl#_one-way].

Using these MAPs, complex MEPs can be created. For example:

1. **Asynchronous MEP**: Using `reply endpoint` MAP, an asynchronous transport may be specified for a synchronous request. For example, a client application might send a request over HTTP and ask to receive the response through SMTP.

2. **Conversation MEP**: Using `relationship` MAP, that defines the relationship between two messages, a conversational MEP can be defined by correlating multiple request/response MEPs. For example a client sending a request to service endpoint receives a response with `wsa:RelatesTo` MAP. The service endpoint may optionally include `wsa:MessageID` in the response. This MAP can then be included by the client in `wsa:RelatesTo` MAP in next request to the service endpoint there by starting a conversation.

3. **Distributed MEP**: Using `reply endpoint` and `fault endpoint` MAP, a different transport/address can be specified for receiving normal and fault responses respectively.

11.2.4. Composability with other WS-* specifications

There are several Web services specification (commonly known as WS-* specs [http://en.wikipedia.org/wiki/WS-*]) that make use of the abstract properties defined by WS-Addressing. For example WS-Metadata Exchange [http://en.wikipedia.org/wiki/WS-MetadataExchange] define a bootstrap mechanism for retrieving metadata before the business message exchange can take place. This mechanism involve sending a WS-Transfer [http://www.w3.org/Submission/WS-Transfer/] request for the retrieval of a resource's representation. A typical request message looks like:

```
<s11:Envelope                                                    ❶
        xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:wsa="http://www.w3.org/2005/08/addressing">  ❷
    <s11:Header>
        <wsa:Action>                                             ❸
            http://schemas.xmlsoap.org/ws/2004/09/transfer/Get
        </wsa:Action>
        <wsa:To>http://example.org/metadata</wsa10:To>
        <wsa:ReplyTo>
            <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous
            </wsa10:Address>
        </wsa:ReplyTo>
        <wsa:MessageID>
            uuid: 68da6b24-7fa1-4da2-8a06-e615bfa3d2d0
        </wsa:MessageID>
    </s11:Header>
    <s11:Body/>
</s11:Envelope>
```

❶ SOAP request message to retrieve metadata about a Web service endpoint.
❷ WS-Addressing namespace URI bound to "`wsa`" prefix.
❸ The standard WS-Addressing MAPs used to convey the semantics (`wsa:Action`), receiver (`wsa:To`) of the message, intended receiver of reply (`wsa:ReplyTo`) message and identity (`wsa:MessageID`) information of the message

This message has an empty SOAP Body and relies completely upon standard MAPs to convey all the information. Similarly, a WS-Metadata Exchange response message with metadata looks like:

```
<s11:Envelope
        xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <s11:Header>
        <wsa:Action>                                                    ❶
            http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse
        </wsa:Action>
        <wsa:RelatesTo>
            uuid: 68da6b24-7fa1-4da2-8a06-e615bfa3d2d0
        </wsa:RelatesTo>
    </s11:Header>
    <s11:Body/>                                                         ❷
    ...
    <s11:Body/>
</s11:Envelope>
```

❶ The standard WS-Addressing MAPs used to convey the semantics (`wsa:Action`) of the response message and relationship (`wsa:RelatesTo`) to the request message.

❷ Abbreviated SOAP Body for simplicity which otherwise would contain the MEX response.

WS-Reliable Messaging [http://en.wikipedia.org/wiki/WS-ReliableMessaging] describes a protocol that allows messages to be delivered reliably between distributed applications in the presence of software component, system or network failures. This specification defines protocol messages that must be exchanged between client and service endpoint, before the business message exchange, in order to deliver the messages reliably. For example, RM Source sends `<CreateSequence>` request message to RM Destination to create an outbound sequence. The message looks like:

```
<s11:Envelope
        xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:wsa=" http://www.w3.org/2005/08/addressing"
        xmlns:wsrm=" http://schemas.xmlsoap.org/ws/2005/02/rm">
    <s11:Body>                                                          ❶
        <wsrm:CreateSequence>
            <wsrm:AcksTo>
                <wsa:Address>
                    http://www.w3.org/2005/08/addressing/anonymous
                </wsa:Address>
            </wsrm:AcksTo>
        </wsrm:CreateSequence>
    </s11:Body>
</s11:Envelope>
```

❶ SOAP Body of the request message.

The Body contains an element, `wsrm:AcksTo` (of the type Endpoint Reference), that specifies the endpoint reference to which `<SequenceAcknowledgement>` messages and faults related to sequence creation are sent.

WS-Secure Conversation, WS-Trust, WS-Policy and other similar specifications use the constructs defined by WS-Addressing as building blocks.

11.3. WS-Addressing Versions

There are two prominent versions of WS-Addressing that are commonly used:

- W3C WS-Addressing [http://www.w3.org/2002/ws/addr/]
- Member Submission WS-Addressing [http://www.w3.org/Submission/ws-addressing/]

Sun, IBM, BEA, Microsoft and SAP co-authored and submitted a WS-Addressing specification to W3C in August 2004. W3C chartered a new Working Group with a mission to produce a W3C Recommendation [http://www.w3.org/2005/10/Process-20051014/tr.html#RecsW3C] for WS-Addressing by refining the submitted specification. The original specification submitted to W3C is referred as "Member Submission WS-Addressing" or "Submission WS-Addressing". The term Member Submission [http://www.w3.org/2005/10/Process-20051014/submission.html#Submission] is defined by W3C.

The WG was chartered [http://www.w3.org/2006/10/ws-addr-charter.html#deliverables] to deliver a W3C Recommendation for WS-Addressing Core [http://www.w3.org/TR/ws-addr-core], SOAP Binding [http://www.w3.org/TR/ws-addr-soap] (mapping abstract properties defined in Core to SOAP 1.1 and 1.2) and WSDL Binding [http://www.w3.org/TR/ws-addr-wsdl] (mechanisms to define property values in WSDL 1.1 and WSDL 2.0 service descriptions) specification. This separate between Core/Bindings is common methodology where Core is relevant to application developers and Binding (both SOAP and WSDL) is relevant for Web service stack implementers. This collective set of specifications is referred as "W3C WS-Addressing".

Eclipse Implementation of XML Web Services supports both versions out-of-the-box. Check below on how to enable either of the versions on a service endpoint starting from Java or starting from WSDL.

11.4. Describing WS-Addressing in WSDL

WS Addressing 1.0- Metadata defines standard ways to describe message addressing properties like Action, Destination in wsdl and also indicate the use of Addressing in wsdl. WS-Addressing Metadata specification replaces the previous Web Services Addressing 1.0 - WSDL Binding [http://www.w3.org/TR/2006/CR-ws-addr-wsdl-20060529/] specification in candidate recommendation earlier. If you are still using wsdls conforming to WS Addressing 1.0 - WSDL Binding specification, skip to the next section. Still you may want to update your wsdls to use in standard ways defined by the W3C recommended Addressing 1.0 - Metadata specification for better interoperability. Also, There is no standard mechanism to describe Member Submission version support in the WSDL and some implementaions have used WS-Policy to indicate the support of member submission version and Eclipse Implementation of XML Web Services understands such assertion.

11.4.1. WS-Addressing 1.0 - Metadata

Addressing 1.0 - Metadata [http://www.w3.org/TR/2007/REC-ws-addr-metadata-20070904/] specification uses Web Services Policy Framework (WS Policy 1.5 [http://www.w3.org/TR/2007/REC-ws-policy-20070904]) and Web Services Policy - Attachment [WS Policy 1.5 - Attachment [http://www.w3.org/TR/2007/REC-ws-policy-attach-20070904]] specifications to express the support of Web Services Addressing 1.0. A new policy assertion `<wsam:Addressing>` is defined to express the support of Addressing. The `wsam:Addressing` policy assertion applies to the endpoint policy subject and may be attached to `wsdl11:port` or `wsdl11:binding`.

Indicating the requirement of WS-Addressing: When

```
<wsam:Addressing>
```

is present in a Policy alternative, one is required to use WS-Addressing to communicate with the subject.

Indicating the support of WS-Addressing:

```
<wsam:Addressing wsp:Optional="true">
```

can be used to indicate support for WS-Addressing but does not require the use of it. In these cases, there are no restrictions about the use of WS-Adressing.

In certain cases, the endpoint can lay some restrictions to indicate the messages it can accept with WS-Addressing. Nested asertions can be used to restrict the use of response endpoint inside the `<wsam:Addressing>` assertion.

Requiring the use of Non-Anonymous response endpoints:

```
<wsam:Addressing>
    <wsp:Policy>
        <wsam:NonAnonymousResponses/>
    </wsp:Policy>
</wsam:Addressing>
```

can be used to indicate that the subject requires WS-Addressing and requires the use of non-anonymous response EPRs. In this case, the response endpoint in the request messages will have to use something other than the anonymous URI as the value of address. This is typically used when the response needs to be sent to a third entity other than the client and service and the response is sent to the non-anonyous URI through a new connection Requiring the use of Anonymous response endpoints:

```
<wsam:Addressing>
    <wsp:Policy>
        <wsam:AnonymousResponses/>
    </wsp:Policy>
</wsam:Addressing>
```

can be used to indicate that the subject requires WS-Addressing and requires the use of anonymous responses. In this case, the endpoint requires request messages to use response endpoint EPRs that contain the anonymous URI ("`http://www.w3.org/2005/08/addressing/anonymous`") or None URI ("`http://www.w3.org/2005/08/addressing/none`") as the value of address.

11.4.2. WS-Addressing WSDL Binding

W3C WS-Addressing WSDL Binding defines an extensibility element, wsaw:UsingAddressing [http://www.w3.org/TR/2006/CR-ws-addr-wsdl-20060529/#uaee], that can be used to indicate that an endpoint conforms to the WS-Addressing specification. Eclipse Implementation of XML Web Services generates this extension element in the WSDL if W3C WS-Addressing is enabled on the server-side. On the client side, the RI recognizes this extension element and enforce the rules defined by the W3C specification. This extensibility element may be augmented with `wsdl:required` attribute to indicate whether WS-Addressing is required (true) or not (false).

W3C WS-Addressing WSDL Binding defines `wsaw:Anonymous` element which when used in conjunction with `wsaw:UsingAddressing` define assertions regarding a requirement or a constraint in the use of anonymous URI in EPRs sent to the endpoint. The WSDL Binding defines three distinct values: `optional`, `required` and `prohibited` to express the assertion. The default value of `wsaw:Anonymous` (equivalent to not present) is `optional`. An operation with `required` `wsaw:Anonymous` value is shown below:

```
<wsaw:UsingAddressing wsdl:required="true"/>
<soap:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document"/>
<operation name="addNumbers">
    <soap:operation soapAction=""/>

    ...

    <wsaw:Anonymous>required</wsaw:Anonymous>
</operation>
<soap:binding>
```

In this case, a message received at the endpoint, for this operation, with a non-anonymous ReplyTo or FaultTo EPR will result in a fault message returned back to the client with `wsa:OnlyAnonymousAddressSupported` fault code. There is no such equivalent feature in Member Submission WS-Addressing.

11.5. Configuring Addressing on Endpoint

This section describes how W3C and Member Submission WS-Addressing can be enabled/disabled on the server-side.

11.5.1. Starting from WSDL

Starting from WSDL, If the wsdl contains the above described metadata to indicate use addressing at endpoint scope, Addressing is enabled on the server-side. See Describing WS-Addressing in WSDL section for more details.

11.5.2. Starting from Java

This section describes how WS-Addressing can be enabled/disabled if you develop an endpoint starting from a Java SEI.

By default, WS-Addressing is disabled on an endpoint starting from Java. If that is the expected behavior, then nothing else needs to be done. In that case any WS-Addressing headers received at the endpoint are treated like SOAP headers targeted for the appliaction and are ignored.

11.5.2.1. Addressing annotations

If WS-Addressing support needs to be enabled on an endpoint, then along with `javax.jws.WebService` annotation, `javax.xml.ws.soap.Addressing` annotation need to be specified for enabling W3C WS-Addressing. If Member Submission WS-Addressing needs to be enabled then `com.sun.xml.ws.developer.MemberSubmissionAddressing` annotation needs to be specified on the service endpoint. For example, the service endpoint in `fromjava-wsaddressing` sample looks like:

```
@javax.xml.ws.soap.Addressing
@javax.jws.WebService
public class AddNumbersImpl {

    // ...

}
```

To enable, Member Submission WS-Addressing, the SEI definition needs to be changed to:

```
@com.sun.xml.ws.developer.MemberSubmissionAddressing
@javax.jws.WebService
public class AddNumbersImpl {

    // ...

}
```

Once WS-Addressing support is enabled on a service endpoint, then:

- In the generated WSDL, corresponding metadata as described in section Describing Addressing in WSDL is generated.

- All WS-Addressing headers are understood, i.e. if any WS-Addressing header is received with a `mustUnderstand="1"`, then a mustUnderstand fault is not thrown back.
- All WS-Addressing headers received at the endpoint are checked for correct syntax, for example an error is returned back if `wsa:ReplyTo` header does not match the infoset defined in the corresponding specification.
- If any WS-Addressing header received at the endpoint is not of correct cardinality, then an error is returned back to the client.
- If `wsa:Action` header value does not match with that expected for that operation, then an error is returned back to the client.
- Any response message sent back to the client contains the required WS-Addressing headers.

11.5.2.2. Is Addressing Optional or Required ?

Both `javax.xml.ws.soap.Addressing` and `com.sun.xml.ws.developer.MemberSubmissionAddressing` annotations take two optional Boolean parameters, `enabled` (default true) and `required` (default false). If `required` is specified true, then WS-Addressing rules are enforced. Otherwise the inbound message is inspected to find out if WS-A is engaged and then the rules are enforced. See When is WS-Addressing engaged? section for more details on enforcement during runtime.

For example, to enforce Member Submission WS-Addressing rules on the server side, the above code sample will change to:

```
@com.sun.xml.ws.developer.MemberSubmissionAddressing(enabled = true,
        required = true)
@javax.jws.WebServicepublic
class AddNumbersImpl {

    // ...

}
```

11.6. On the client side

This section describes how WS-Addressing can be enabled/disabled on the client-side. Eclipse Implementation of XML Web Services follows the standard extensibility elements in WSDL to enable WS-Addressing support on the client side. In addition, it also allows the client to instruct Eclipse Implementation of XML Web Services to disable WS-Addressing processing. The assumption is that in this case the client has it's own WS-Addressing processing module. For example, a Dispatch-based client in MESSAGE mode may be used to perform non-anonymous ReplyTo/FaultTo processing.

11.6.1. Implicit behavior

As defined in Describing WS-Addressing in WSDL, If the WSDL contains metadata about the support or requirement of WS-Addressing, Eclipse Implementation of XML Web Services runtime enables Addressing feature on the client-side.

- Generates `Action`, `To`, `MessageID` and anonymous `ReplyTo` headers on the outbound request.
- Any WS-Addressing headers received on the client are processed.

There is no standard extensibility element for Member Submission WS-Addressing and so there is no implicit behavior defined. It can only be explicitly enabled as described in the next section.

11.6.2. Explicit enabling

If a WSDL does not contain WS-Addressing standard extensibility element, then either W3C WS-Addressing or Member Submission WS-Addressing can be explicitly enabled using `createDispatch` and `getPort` methods on `javax.xml.ws.Service`. The following new APIs are added in Jakarta XML Web Services 2.1:

- `<T> Dispatch<T> createDispatch(javax.xml.namespace.QName portName, java.lang.Class<T> type, Service.Mode mode, WebServiceFeature... features)`
- `Dispatch<java.lang.Object> createDispatch(javax.xml.namespace.QName portName, javax.xml.bind.JAXBContext context, Service.Mode mode, WebServiceFeature... features)`
- `<T> T getPort(java.lang.Class<T> serviceEndpointInterface, WebServiceFeature... features)`
- `<T> T getPort(javax.xml.namespace.QName portName, java.lang.Class<T> serviceEndpointInterface, WebServiceFeature... features)`

Each method is a variation of an already existing method in Jakarta XML Web Services. The only addition is an extra var-arg `javax.xml.ws.WebServiceFeature` parameter. A `WebServiceFeature` is a new class introduced in Jakarta XML Web Services 2.1 specification used to represent a feature that can be enabled or disabled for a Web service.

The Jakarta XML Web Services 2.1 specification defines `javax.xml.ws.soap.AddressingFeature` to enable W3C WS-Addressing on the client side. In addition, the Eclipse Implementation of XML Web Services also defines `com.sun.xml.ws.developer.MemberSubmissionAddressingFeature` to enable Member Submission WS-Addressing on the client side.

For example in `fromjava-wsaddressing` example, in order to enable W3C WS-Addressing on a proxy, **wsimport** is used to generate the `AddNumbersImplService` class. Then a port can be obtained using the `getAddNumbersImplPort` method and passing an instance of `javax.xml.ws.AddressingFeature`. The code looks like:

```
new AddNumbersImplService().getAddNumbersImplPort(new
 javax.xml.ws.AddressingFeature());
```

Similarly, a `Dispatch` instance with Member Submission WS-Addressing can be created as:

```
new AddNumbersImplService().createDispatch(
        new QName("http://server.fromjava_wsaddressing/",
                "AddNumbersImplPort"),
        SOAPMessage.class,
        Service.Mode.MESSAGE,
        new com.sun.xml.ws.developer.MemberSubmissionAddressingFeature());
```

Feature Parameters

Both `javax.xml.ws.soap.AddressingFeature` and `com.sun.xml.ws.developer.MemberSubmissionAddressingFeature` take two optional Boolean parameters, `enabled` (default true) and `required` (default false). If enabled, all WS-Addressing headers are generated for an outbound message. If `required` is specified true, then WS-Addressing rules are enforced for inbound message. Otherwise the inbound message is inspected to find out if WS-A is engaged and then the rules are enforced.

For example, to enforce Member Submission WS-Addressing rules on the client side, the above code sample will change to:

```
new AddNumbersImplService().getAddNumbersImplPort(new com.sun.xml
        .ws.developer.MemberSubmissionAddressingFeature(true, true));
```

11.6.3. Explicit disabling

A client may like to instruct Eclipse Implementation of XML Web Services to disable WS-Addressing processing. The assumption is that in this case the client has it's own WS-Addressing processing module. For example, a Dispatch-based client in MESSAGE mode may be used to perform non-anonymous ReplyTo/FaultTo processing.

WS-Addressing processing can be explicitly disabled using one of new methods added to Jakarta XML Web Services 2.1 specification as defined in Section 3.2. For example, W3C WS-Addressing processing can be disabled using the following code:

```
new AddNumbersImplService().getAddNumbersImplPort(new
 javax.xml.ws.AddressingFeature(false));
```

11.7. When is WS-Addressing engaged?

W3C WS-Addressing SOAP Binding defines [http://www.w3.org/TR/2006/REC-ws-addr-soap-20060509/#conformance] that if a receiver processes a message containing a `wsa:Action` header, then SOAP Binding is engaged, and the rules of the specification are enforced. In Eclipse Implementation of XML Web Services, if WS-Addressing is explicitly disabled then the RI does not follow the rules of engagement. However if WS-Addressing is either implicitly or explicitly enabled then Eclipse Implementation of XML Web Services engages WS-Addressing based upon the presence of `wsa:Action` header. Eclipse Implementation of XML Web Services follows same rule for Member Submission version as well.

In effect, if an endpoint advertises WS-Addressing is required in the WSDL and a client does not send any WS-Addressing header then no WS-Addressing fault is returned back to the client. However if the client send `wsa:Action` header then the endpoint will enforce all the rules of the specification. For example, if the `wsa:MessageID` header is missing for a request/response MEP then a fault with appropriate code and sub-code is thrown back to the client.

11.8. Associating Action with an operation

11.8.1. Implicit Action

In most common cases, an implicit Action association, as defined by W3C WS-Addressing 1.0 - Metadata [http://www.w3.org/TR/2007/REC-ws-addr-metadata-20070904/#actioninwsdl] and Member Submission [http://www.w3.org/Submission/ws-addressing/#_Toc77464327], will be sufficient. For such cases, only using the correct annotation to enable Addressing is required. The client looking at such a WSDL will send the implicit `wsa:Action` header. If only Addressing is enabled by using the appropriate annotation at the SEI,

11.8.2. Explicit Action

This section describes how an explicit Action Message Addressing Property can be associated with an operation in the SEI.

W3C WS-Addressing W3C WS-Addressing 1.0 - Metadata [http://www.w3.org/TR/2007/REC-ws-addr-metadata-20070904/#actioninwsdl] and Member Submission WS-Addressing [http://www.w3.org/Submission/ws-addressing/#_Toc77464325] define mechanisms to associate Action Message Addressing Property with an operation. Jakarta XML Web Services defines `javax.xml.ws.Action` and `javax.xml.ws.FaultAction` annotations to explicitly associate an Action with `input`, `output`,

and `fault` messages of the mapped WSDL operation. For example, one of the methods in the `fromjava-wsaddressing` sample looks like:

```
@Action(input = "http://example.com/input3", output = "http://example.com/
output3",
        fault = {@FaultAction(className = AddNumbersException.class,
        value = "http://example.com/fault3")})
public int addNumbers3(int number1, int number2) throws AddNumbersException {

    // ...

}
```

The generated WSDL fragment looks like:

```
<operation name="addNumbers3">
    <input wsam:Action="http://example.com/input3"
            message="tns:addNumbers3"/>
    <output wsam:Action="http://example.com/output3"
            message="tns:addNumbers3Response"/>
    <fault message="tns:AddNumbersException" name="AddNumbersException"
            wsam:Action="http://example.com/fault3"/>
</operation>
```

where `wsam` is bound to W3C WS-Addressing 1.0 - Metadata namespace or Member Submission namespace depending upon the annotation used to enable Addressing.

12. Stateful Webservice

12.1. Introduction

Eclipse Implementation of XML Web Services has a vendor extension that allows developers to bring back object state to the web service world. Normally, Eclipse Implementation of XML Web Services only creates one instance of a service class, and have it serve all incoming requests concurrently. This makes it essentially impossible to use instance fields of the service class for any meaningful purpose.

Because of this, people end up coding like C, in anti-OO fashion. Either that or you end up with writing boiler-plate code to dispatch a request to the right instance of your real domain object.

The stateful web service support in Eclipse Implementation of XML Web Services resolves this problem by having Eclipse Implementation of XML Web Services maintain multiple instances of a service. By using WS-Addressing behind the scene, it provides a standard-based on-the-wire protocol and easy-to-use programming model.

12.2. Usage

Application service implementation classes (or providers) who'd like to use the stateful web service support must declare `@Stateful` annotation on a class. It should also have a **public static** method/field that takes `StatefulWebServiceManager`.

```
@Stateful
@WebService
@Addressing
class BankAccount {

    protected final int id;
    private int balance;
```

```
    BankAccount(int id) {
        this.id = id;
    }

    @WebMethod
    public synchronized void deposit(int amount) {
        balance += amount;
    }

    // either via a public static field

    public static StatefulWebServiceManager<BankAccount> manager;

    // ... or  via a public static method (the method name could be
    // anything)

    public static void setManager(StatefulWebServiceManager<BankAccount>
                                            manager) {
        // ...
    }

}
```

After your service is deployed but before you receive a first request, the resource injection occurs on the field or the method.

A stateful web service class does not need to have a default constructor. In fact, most of the time you want to define a constructor that takes some arguments, so that each instance carries certain state (as illustrated in the above example).

Each instance of a stateful web service class is identified by an unique `EndpointReference`. Your application creates an instance of a class, then you'll have Eclipse Implementation of XML Web Services assign this unique EPR for the instance as follows:

```
@WebService
class Bank { // this is ordinary stateless service

    @WebMethod
    public synchronized W3CEndpointReference login(int accountId,
                                                   int pin) {
        if (!checkPin(pin))
            throw new AuthenticationFailedException("invalid pin");
        BankAccount acc = new BankAccount(accountId);
        return BankAccount.manager.export(acc);
    }

}
```

Typically you then pass this EPR to remote systems. When they send messages to this EPR, Eclipse Implementation of XML Web Services makes sure that the particular exported instance associated with that EPR will receive a service invocation.

12.3. Things To Consider

When you no longer need to tie an instance to the EPR, use `unexport(Object)` so that the object can be GC-ed (or else you'll leak memory). You may choose to do so explicitly, or you can rely on the time out by using `setTimeout(long, Callback)`.

`StatefulWebServiceManager` is thread-safe. It can be safely invoked from multiple threads concurrently.

13. Catalog

13.1. Catalog Support

Often times, such as for performance reason or your application specific needs, you have a need where you want to resolve the WSDL/Schema documents resolved from the copy of it you have either bundled with your client or server or just to resolve it not from where a WSDL/schema imports points to but rather from where you want it to be picked up.

13.1.1. A Sample catalog file

Example 36. `jax-ws-catalog.xml`

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
        prefer=" system">

    ...

    < system systemId=" http://foo.org/hello?wsdl" uri="HelloService.wsdl"/>
</catalog>
```

13.1.1.1. Location of the catalog file

- For **wsimport** command-line or ant task
  - use `-catalog` option to pass the catalog file. The name of the catalog file does not matter for **wsimport** tool but for consistency you may like to call it `jax-ws-catalog.xml`
  - use

    ```
    <wsimport catalog="''>
    ```

    or

    ```
    <xmlcatalog ... />
    ```

    with **wsimport** ant task
- Client Runtime
  - `META-INF/jax-ws-catalog.xml` picked up from classpath
- Lightweight HTTP server (j2se) based endpoints
  - `META-INF/jax-ws-catalog.xml` picked up from classpath
- Servlet based endpoints or Jakarta Enterprise Web Services based Web Module
  - `WEB-INF/jax-ws-catalog.xml`
- Jakarta Enterprise Web Services based EJB Modules
  - `META-INF/jax-ws-catalog.xml`

For details on XML catalog see here [http://xml.apache.org/commons/components/resolver/resolver-article.html].

14. WAR File Packaging

14.1. The WAR Contents

Typically, one creates the WAR file with a GUI development tool or with the `ant war` task from the generated artifacts from **wsimport**, **wsgen**, or **annotationProcessing** tools.

For example, a sample WAR file starting from a WSDL file:

Table 26. Files contained in WAR when starting from WSDL

File	Description
`WEB-INF/classes/hello/HelloIF.class`	SEI
`WEB-INF/classes/hello/HelloImpl.class`	Endpoint
`WEB-INF/sun-jaxws.xml`	Eclipse Implementation of XML Web Services deployment descriptor
`WEB-INF/web.xml`	Web deployment descriptor
`WEB-INF/wsdl/HelloService.wsdl`	WSDL
`WEB-INF/wsdl/schema.xsd`	WSDL imports this Schema

14.2. The `sun-jaxws.xml` File

The `<endpoints>` element contain one or more `<endpoint>` elements. Each endpoint represents a port in the WSDL and it contains all information about `implementation` class, servlet `url-pattern`, `binding`, `WSDL`, `service`, `port` QNames. The following shows a `sun-jaxws.xml` file for a simple `HelloWorld` service. `sun-jaxws.xml` is the schema instance of sun-jaxws.xsd.

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
        version="2.0">
    <endpoint name="MyHello" implementation="hello.HelloImpl"
            url-pattern="/hello"/>
</endpoints>
```

Endpoint can have the following attributes:

Table 27. `sun-jaxws.xml` - Endpoint element attributes.

Attribute	Optional	Use
`name`	N	Name of the endpoint
`wsdl`	Y	Primary wsdl file location in the WAR file. For e.g. `WEB-INF/wsdl/HelloService.wsdl`. If

Attribute	Optional	Use
		this isn't specified, Eclipse Implementation of XML Web Services will create and publish a new WSDL. When the service is developed from Java, it is recommended to omit this attribute.
`service`	Y	QName of WSDL service. For e.g. `{http://example.org/}HelloService`. When the service is developed from Java, it is recommended to omit this attribute.
`port`	Y	QName of WSDL port. For e.g. `{http://example.org/}HelloPort`. When the service is developed from Java, it is recommended to omit this attribute.
`implementation`	N	Endpoint implementation class name. For e.g: `hello.HelloImpl`. The class should have a `@WebService` annotation. Provider based implementation class should have a `@WebServiceProvider` annotation.
`url-pattern`	N	Should match `<url-pattern>` in `web.xml`
`binding`	Y	Binding id defined in the Jakarta XML Web Services API. The possible values are: • `"http://schemas.xmlsoap.org/wsdl/soap/http"` • `"http://www.w3.org/2003/05/soap/bindings/HTTP/"` If omitted, it is considered `SOAP1.1` binding.
`enable-mtom`	Y	Enables MTOM optimization. `true` or `false`. Default is `false`.

Endpoint can have a optional `handler-chain` element:

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

```
        <handler-chain>
            <handler-chain-name>somename</handler-chain-name>
            <handler>
                <handler-name>MyHandler</handler-name>
                <handler-class>hello.MyHandler</handler-class>
            </handler>
        </handler-chain>
    </endpoint>
</endpoints>
```

14.3. The `web.xml` File

The following shows a `web.xml` file for a simple `HelloWorld` service. It specifies Eclipse Implementation of XML Web Services specific listener, servlet classes. These classes are `com.sun.ws.transport.http.servlet.WSServletContextListener`, and `com.sun.xml.ws.transport.http.servlet.WSServlet` is servlet

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
        PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
        "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
    <listener>
        <listener-class>
            com.sun.xml.ws.transport.http.servlet.WSServletContextListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>hello</servlet-name>
        <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>hello</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>60</session-timeout>
    </session-config>
</web-app>
```

Remember these requirements when building a WAR:

- WSDL and auxiliary WSDL, Schema files should be packaged under `WEB-INF/wsdl` dir. It is recommended that they need not be packaged when the service is started from Java
- WebService implementation class should contain a `@WebService` annotation. Provider based endpoints should have a `@WebServiceProvider` annotation.
- `wsdl`, `service`, `port` attributes are mandatory for `Provider` based endpoints and can be specified in the `@WebServiceProvider` annotation or deployment descriptor (`sun-jaxws.xml`).

15. Interoperability

Please refer to Metro User's Guide [https://javaee.github.io/metro-jax-ws/doc/user-guide/] for interoperability capabilities.

16. Endpoint API

Web Service endpoints can be created and published programmatically using `javax.xml.ws.Endpoint` API. An endpoint consists of a Web Service Implementation object and some configuration information. The implementation hosts the web service endpoint using a light weight http server and clients can access the web service as if the endpoint is deployed in a J2EE container. This means that there is no need to have any J2EE servlet or EJB container to host the endpoint. The `Endpoint` API provides a way to configure the endpoint with the necessary binding, metadata (WSDL and schema documents), handlers etc.

16.1. `Endpoint`

An endpoint can be created using any of the following constructors:

```
Endpoint.create(implementor)

Endpoint.create(bindingId,implementor)

Endpoint.publish(address, implementor)
```

Once the `Endpoint` object is created using the first two constructors, it can be published using `Endpoint.publish()`. Any published `Endpoint` can be stopped using `Endpoint.stop()`. `samples/supplychain/src/supplychain/server/WarehouseLightWeight.java` supplychain sample shows creating and publishing an `Endpoint`.

16.2. `Endpoint` and `Properties`

An endpoint can be configured to match service name and port name of WSDL using properties. This overwrites implementor object's serviceName, portName from `@WebService` annotation. The port address for an endpoint is patched only if the corresponding port's service name, and port name in WSDL are matched.

Example 37. `Endpoint` and `Properties` Example

```
Endpoint endpoint = ...

Map<String, Object> map = new HashMap<String, Object>();
map.put(Endpoint.WSDL_SERVICE, new QName(...));
map.put(Endpoint.WSDL_PORT, new QName(...));

endpoint.setProperties(map);
```

16.3. `Endpoint` and `Binding`

An endpoint can be configured for different bindings using binding ids. These binding ids are defined in Jakarta XML Web Services API and endpoint can be configured by specifying `@BindingType` annotation or using binding id in the `Endpoint()` constructors. The parameter in constructor overwrites binding defined by `@BindingType` annotation. If the binding is not specified using `@BindingType` or using a parameter in `Endpoint()` constructor, the default binding is `SOAP1.1/HTTP`. Binding object is used to configure MTOM, handler chain etc. SOAP binding object is used to configure SOAP binding specifics like roles.

For example:

Example 38. `Endpoint` and `Binding` Example

The following configures the endpoint for `XML/HTTP` binding.

```
Endpoint endpoint = Endpoint.create(HTTPBinding.HTTP_BINDING,
        implementor);
```

Working with a `Binding` object:

```
// setting MTOM
SOAPBinding binding = (SOAPBinding) endpoint.getBinding();
binding.setMTOMEnabled(true);

// setting SOAP binding roles
binding.setRoles(...);

// setting handler chain
binding.setHandlerChain(...);
```

16.4. `Endpoint` and `metadata`

When the service endpoint is created using existing java classes, the implementation dynamically generates and publishes WSDL and schema documents. But when the service endpoint is created using existing WSDL documents, the same WSDL documents can be used for publishing using metadata facility. When a `Source` object is created, set `systemId` always and make sure the imports are resolvable w.r.t `systemId`s.

Example 39. `Endpoint` and `metadata` Example

```
// metadata processing for WSDL, schema files
List<File> metadataFile =...
List<Source> metadata = new ArrayList<Source>();
for (File file : metadataFile) {
    Source source = new StreamSource(new FileInputStream(file));
    source.setSystemId(file.toURL().toExternalForm());
    metadata.add(source);
}
endpoint.setMetadata(metadata);
```

17. Modular Databinding

17.1. Introduction

The Eclipse Implementation of XML Web Services used to be dependent on the Eclipse Implementation of JAXB for databinding. Jakarta XML Binding and Jakarta XML Web Services implementations have been decoupled, and databinding is now modular. The EclipseLink Jakarta XML Binding implementation, plus EclipseLink extensions, is called MOXy. The `org.eclipse.persistence.moxy.jar` file is bundled with GlassFish Server, which supports the Eclipse Implementation of JAXB and MOXy as databinding providers. For standalone distributions, databinding plugins can be found in `lib/plugins` folder in the distribution. The MOXy implementation (library) is not bundled with Eclipse Implementation of XML Web Services. It's expected from user to provide MOXy jars to classpath whenever MOXy databinding is required. EclipseLink Jakarta XML Binding compiler is not included as well, but can be used with GlassFish Server. Download the EclipseLink zip file at http://www.eclipse.org/eclipselink/downloads/ and unzip it.

17.2. Configure databinding for JVM

To specify the databinding provider for the JVM, set the `com.sun.xml.ws.spi.db.BindingContextFactory` JVM property to one of the following values:

- **`com.sun.xml.ws.db.glassfish.JAXBRIContextFactory`** Specifies the Jakarta XML Binding reference implementation. This is the default.
- **`com.sun.xml.ws.db.toplink.JAXBContextFactory`** Specifies the EclipseLink MOXy Jakarta XML Binding binding.

For example:

```
asadmin create-jvm-options -
Dcom.sun.xml.ws.spi.db.BindingContextFactory=com.sun.xml.ws.db.toplink.JAXBContext
```

17.3. Configure databinding for an endpoint

To specify the databinding provider for a web service endpoint:

- Set the `com.oracle.webservices.api.databinding.DatabindingModeFeature` feature during `WebServiceFeature` list initialization or using the `add` method. Allowed values are as follows:
  - **`com.oracle.webservices.api.databinding.DatabindingModeFeature.GLASSFISH_JAXB`** Specifies the Eclipse Implementation of JAXB. This is the default.
  - **`com.sun.xml.ws.db.toplink.JAXBContextFactory.ECLIPSELINK_JAXB`** Specifies Eclipselink MOXy Jakarta XML Binding binding.

  For example:

```
import javax.xml.ws.WebServiceFeature;
import com.oracle.webservices.api.databinding.DatabindingModeFeature;
import com.sun.xml.ws.db.toplink.JAXBContextFactory;
...
WebServiceFeature[] features = { new
 DatabindingModeFeature(JAXBContextFactory.ECLIPSELINK_JAXB)};
```

- Set the `com.oracle.webservices.api.databinding.DatabindingModeFeature` feature using the `@DatabindingMode` annotation. For example:

```
import javax.jws.WebService;
import com.oracle.webservices.api.databinding.DatabindingMode;
import com.sun.xml.ws.db.toplink.JAXBContextFactory;
...
@WebService
@DatabindingMode(JAXBContextFactory.ECLIPSELINK_JAXB);
```

- Set the `databinding` attribute of the endpoint element in the `sun-jaxws.xml` file. Allowed values are `glassfish.jaxb` or `eclipselink.jaxb`. For example:

```
<endpoint
```

```
    name='hello'
    implementation='hello.HelloImpl'
    url-pattern='/hello'
    databinding='eclipselink.jaxb'/>
```

18. External Web Service Metadata

It's one of goals of Eclipse Implementation of XML Web Services to make development of web services as simple as possible, so using of java annotations perfectly makes sense. However there are usecases where it is impossible to use them. For example if we need to expose existing component as a web service, but we have no source code, just binaries. In such scenarios we need not to rely on annotations and Eclipse Implementation of XML Web Services framework needs a different way how to obtain necessary metadata. The solution is to provide metadata in xml files and to configure Eclipse Implementation of XML Web Services framework in a way it's aware of them.

18.1. Configuration files

To specify classes' metadata externally, each java class requires separate file. The way how Eclipse Implementation of XML Web Services framework handles this metadata depends on attributes of xml root element `java-wsdl-mapping`:

Example 40. `webservices.war/WEB-INF/classes/external-metadata.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<java-wsdl-mapping xmlns="http://xmlns.oracle.com/webservices/jaxws-
databinding"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.oracle.com/webservices/jaxws-
databinding jaxws-ri-external-metadata.xsd"
       java-type-name="org.example.BlackBoxImpl"
       existing-annotations="ignore"
       databinding="glassfish.jaxb">

    <web-service name="" target-namespace="mynamespace"/>

</java-wsdl-mapping>
```

- `java-type-name="org.example.BlackBoxImpl"`

  attribute defines what class is the definition file for. Having this information in xml file allows us to provide just list of xml files and framework itself knows what to do with those.

- `existing-annotations="ignore"`

  This attribute says if and how should be java annotations found in the java class handled. Possible values are:

  - `ignore`

    - annotations found in java class are ignored; Eclipse Implementation of XML Web Services framework behaves as if there was no other metadata than one in xml file

  - `merge`

- annotations found in java class are considered, but metadata in xml file is overriding them; if an annotation is present in java file only, framework uses it, if found in both annotation and xml file, the latter one is hiding the first one.

18.2. XSD Schema

As you probably noticed in an example above, there is a new schema for configuration files: http://xmlns.oracle.com/webservices/jaxws-databinding [http://xmlns.oracle.com/webservices/jaxws-databinding/1.0/jaxws-databinding.xsd]. The schema is designed to reflect Web Service Metadata (JSR-181) and Jakarta XML Web Services Annotations (JSR-224) so structure should be really intuitive to developers - see following table:

Table 28. Mapping java `@Annotation`-s to xml elements examples

@Annotation	corresponding xml element
`javax.jws.WebService(name="AddNumbers", portName="AddNumbersPort")`	`<web-service name="AddNumbers" port-name="AddNumbersPort" />`
`javax.xml.ws.WebServiceProvider(targetNamespace="urn:test", serviceName="Endpoint", portName="EndpointPort")`	`<web-service-provider target-namespace="urn:test" service-name="Endpoint" port-name="EndpointPort" />`
`javax.xml.ws.ServiceMode(Service.Mode.MESSAGE)`	`<service-mode value="MESSAGE" />`

18.3. Passing Configuration Files to Eclipse Implementation of XML Web Services

There are different stages where we need to pass the collection of files to Eclipse Implementation of XML Web Services framework:

- wsgen: when starting "from java" - better to say if we have no wsdl prepared, we need to pass the classpath to implementations to be used together with a list of external metadata files to wsgen. Therefore wsgen tool has a new option `-x <path>`. If there are several such files, the option must be repeated before each path. Corresponding ant task supports new nested element "external-metadata", see following example:

Example 41. wsgen ant task example: `build.xml`

```
<wsgen  sei="org.example.server.BlackboxService"
                destdir="${build.classes.home}"
                resourcedestdir="${build.classes.home}"
                sourcedestdir="${build.classes.home}"
                keep="true"
                verbose="true"
                genwsdl="true"
                protocol="soap1.1"
                extension="false"
                inlineSchemas="false">

            <externalmetadata file="${basedir}/etc/external-metadata.xml" /
>
            <classpath>
                <pathelement location="${build.classes.home}"/>
```

```
            </classpath>

        </wsgen>
```

- wsimport - if you start from wsdl, no extra parameter is necessary - artifacts are generated on wsdl only and external metadata are required later, in runtime.

- runtime - in runtime, it's necessary to tell somehow to container (Servlet or JEE) what xml files to load. Currently, Eclipse Implementation of XML Web Services Servlet deployment is supported - `sun-jaxws.xml` schema has been updated to support new elements - `<external-metadata>`, saying to a container to parse the resources when doing a deployment:

Example 42. `webservice-module.war/WEB-INF/sun-jaxws.xml`

```
<?xml encoding="UTF-8" version="1.0" ?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
 version="2.0">
    <endpoint implementation="org.example.server.BlackboxService" url-
pattern="/WS" name="WS">
        <external-metadata resource="external-metadata.xml" />
    </endpoint>
</endpoints>
```

For more complete example see samples.

Tools

Table of Contents

1. Overview

1.1. How do I pick a tool?

The following lists the process to create a web service starting from Java sources, classes, and a WSDL file (server side):

1. Starting from Java sources:

    a. Use **annotationProcessing** Ant task to generate the artifacts required by the Jakarta XML Web Services specification.

    b. Package the `web.xml`, `sun-jaxws.xml`, service endpoint interface and implementation class, value types, and generated classes, if any, into a WAR file,

c. Deploy the WAR to a web container.

2. Starting from Java classes:

   a. Use **wsgen** to generate portable artifacts.

   b. Package the `web.xml`, `sun-jaxws.xml`, service endpoint interface and implementation class, value types, and generated classes, if any, into a WAR file,

   c. Deploy the WAR to a web container.

3. Starting from a WSDL file:

   a. Use **wsimport** to generate portable artifacts.

   b. Implement the service endpoint.

   c. Package the WSDL file, schema documents, `web.xml`, `sun-jaxws.xml`, service endpoint interface and implementation class, value types, and generated classes, if any, into a WAR file.

   d. Deploy the WAR to a web container.

The following lists the process to invoke a web service (client side):

1. Starting from deployed web service's WSDL

   a. Use **wsimport** to generate the client-side artifacts.

   b. Implement the client to invoke the web service.

1.2. Running tools on Java SE 8

- Copy `JAXWS_HOME/lib/jakarta.annotation-api.jar` `JAXWS_HOME/lib/jakarta.jws-api.jar` `JAXWS_HOME/lib/jakarta.xml.ws-api.jar` `JAXWS_HOME/lib/jakarta.xml.bind-api.jar` to `$JDK8_HOME/jre/lib/endorsed` directory

For details see Endorsed Directory Mechanism [https://docs.oracle.com/javase/8/docs/technotes/guides/standards/]. Above, `JAXWS_HOME` points to the root directory of the extracted Eclipse Implementation of XML Web Services bundle. `JDK8_HOME` points to JDK 8 installation directory.

1.3. Maven plugins

You can use all jaxws tools within your maven build. Information about maven plugin and its usage is located at the plugin homepage. [https://github.com/eclipse-ee4j/metro-jaxws-commons/tree/master/jaxws-maven-plugin]

2. wsimport

2.1. wsimport Overview

The **wsimport** tool generates Jakarta XML Web Services portable artifacts, such as:

- Service Endpoint Interface (SEI)
- Service

- Exception class mapped from `wsdl:fault` (if any)
- Async Reponse Bean derived from response `wsdl:message` (if any)
- Jakarta XML Binding generated value types (mapped java classes from schema types)

These artifacts can be packaged in a WAR file with the WSDL and schema documents along with the endpoint implementation to be deployed. Eclipse Implementation of XML Web Services 2.3.3 also provides a **wsimport** Ant Task.

2.2. Launching wsimport

- **Solaris/Linux**
  - `JAXWS_HOME/bin/wsimport.sh -help`
- **Windows**
  - `JAXWS_HOME\bin\wsimport.bat -help`

2.3. wsimport Syntax

`wsimport` [OPTION]... <WSDL>

The following table lists the **wsimport** options:

Option	Description
`-d <directory>`	Specify where to place generated output files.
`-classpath <path>`	Specify where to find user class files and wsimport extensions.
`-cp <path>`	Specify where to find user class files and wsimport extensions.
`-b <path>`	Specify external Jakarta XML Web Services or Jakarta XML Binding binding files or additional schema files (Each `<file>` must have its own `-b`).
`-B <jaxbOption>`	Pass this option to Jakarta XML Binding schema compiler.
`-catalog`	Specify catalog file to resolve external entity references, it supports TR9401, XCatalog, and OASIS XML Catalog format. Please read the documentation of Catalog and see catalog sample.
`-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 an HTTP proxy server (port defaults to 8080).
`-J<javacOption>`	Pass this option to Javac compiler. Note: use '=' instead of space as a delimiter between option name and its value.

Option	Description
`-keep`	Keep generated source code files. It is enabled when `-s` option.
`-p <pkg>`	Specifying a target package via this command-line option, overrides any wsdl and schema binding customization for package name and the default package name algorithm defined in the specification.
`-s <directory>`	Specify where to place generated source code files. keep is turned on with this option.
`-encoding <encoding>`	Set the encoding name for generated sources, such as EUC-JP or UTF-8. If `-encoding` is not specified, the platform default encoding is used.
`-verbose`	Output messages about what the compiler is doing.
`-version`	Print version information.
`-fullversion`	Print full version information.
`-clientjar <jarfile>`	Creates the jar file of the generated artifacts along with the WSDL metadata required for invoking the web service.
`-wsdllocation <location>`	`@WebServiceClient.wsdlLocation` value.
`-target <version>`	Generate code as per the given Jakarta XML Web Services specification version. For example, "`-target 2.0`" generates compliant code for Jakarta XML Web Services 2.0 spec. Default value is 2.2.
`-quiet`	Suppress **wsimport** output.
`-XadditionalHeaders`	Map the headers not bound to request or response message to Java method parameters.
`-Xauthfile`	File to carry authorization information in the format `http://username:password@example.org/stock?wsdl`. The asterisk character ("*") can be used to match multiple URL patterns. Default value is `$HOME/.metro/auth`
`-Xdebug`	Print debug information.
`-XdisableAuthenticator`	Disables Authenticator used by Eclipse Implementation of XML Web Services, `-Xauthfile` option will be ignored if `-XdisableAuthenticator` is set.
`-Xno-addressing-databinding`	Enable binding of W3C `EndpointReferenceType` to Java.
`-Xnocompile`	Do not compile generated Java files.
`-XdisableSSLHostnameVerification`	Disbales the SSL Hostname verification while fetching the wsdls.

Multiple Jakarta XML Web Services and Jakarta XML Binding binding files can be specified using `-b` option and they can be used to customize various things like package names, bean names, etc. More

information on Jakarta XML Web Services and Jakarta XML Binding binding files can be found in the WSDL Customization.

2.4. wsimport Example

```
wsimport -p stockquote http://stockquote.xyz/quote?wsdl
```

This will generate the Java artifacts and compile them by importing the http://stockquote.xyz/quote?wsdl [http://stockquote.org/quote?wsdl].

3. wsimport Ant Task

3.1. wsimport Task Overview

The **wsimport** generates Jakarta XML Web Services portable artifacts, such as:

- Service Endpoint Interface (SEI)
- Service
- Exception class mapped from `wsdl:fault` (if any)
- Async Reponse Bean derived from response `wsdl:message` (if any)
- Jakarta XML Binding generated value types (mapped java classes from schema types)

3.2. Using wsimport Task

To use this **wsimport** task, a `<taskdef>` element needs to be added to the project as given below:

```
<taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport">
    <classpath path="jaxws.classpath"/>
</taskdef>
```

where `jaxws.classpath` is a reference to a path-like structure [http://ant.apache.org/manual/using.html#path], defined elsewhere in the build environment, and contains the list of classes required by the Jakarta XML Web Services tools.

3.2.1. Environment Variables

- ANT_OPTS [http://wiki.apache.org/ant/TheElementsOfAntStyle] - command-line arguments that should be passed to the JVM. For example, you can define system properties or set the maximum Java heap size here.

3.2.2. wsimport Task Attributes

wsimport supports the following parameter attributes:

```
<wsimport
        wsdl="..."
        destdir="directory for generated class files"
        sourcedestdir="directory for generated source files"
        keep="true|false"
        encoding="..."
        extension="true|false"
        verbose="true|false"
        wsdlLocation="..."
        clientJar="jar file"
```

```
        catalog="catalog file"
        package="package name"
        target="target release"
        binding="..."
        quiet="true|false"
        fork="true|false"
        failonerror="true|false"
        xadditionalHeaders="true|false"
        xauthfile="authorization file"
        xdisableAuthenticator="true|false"
        xdebug="true|false"
        xNoAddressingDatabinding="true|false"
        xnocompile="true|false">
    <binding dir="..." includes="..."/>
    <arg value="..."/>
    <xjcarg value="..."/>
    <javacarg value="..."/>
    <jvmarg value="..."/>
    <xmlcatalog refid="another catalog file"/>
    <depends file="..."/>
    <produces dir="..."/>
</wsimport>
```

Attribute	Description	Required	Command line
`wsdl`	WSDL file.	Yes.	WSDL location
`destdir`	Specify where to place output generated classes.	No. Defaults to current working directory.	`-d`
`sourcedestdir`	Specify where to place generated source code files, keep is turned on with this option.	No. Defaults to current working directory.	`-s`
`encoding`	Set the encoding name for generated sources, such as UTF-8.	No. Defaults to platform default.	`-encoding`
`keep`	Keep generated source code files, turned on with `sourcedestdir` option.	No. Defaults to false.	`-keep`
`verbose`	Output Eclipse Implementation of XML Web Services version and messages about what the compiler is doing	No. Defaults to false.	`-verbose.`
`binding`	Specify external Jakarta XML Web Services or Jakarta XML Binding binding files or additional schema files (Each `<file>` must have its own `-b`).	No. No defaults.	`-b`
`extension`	allow vendor extensions (functionality not specified by the specification).	No. Defaults to false.	`-extension`

Attribute	Description	Required	Command line
	Use of extensions may result in applications that are not portable or may not interoperate with other implementations.		
`wsdllocation`	The wsdl URI passed thru this option will be used to set the value of `@WebService.wsdlLocation` and `@WebServiceClient.wsdlLocation` annotation elements on the generated SEI and Service interface.	No. Defaults to the wsdl URL passed to wsdl attribute.	`-wsdllocation`
`clientJar`	Creates the jar file of the generated artifacts along with the WSDL metadata required for invoking the web service.	No.	`-clientJar`
`catalog`	Specify catalog file to resolve external entity references, it supports TR9401, XCatalog, and OASIS XML Catalog format. Additionally, ant xmlcatalog type can be used to resolve entities, see **catalog** sample and Catalog.	No. No defaults.	`-catalog`
`package`	Specifies the target package.	No. It default to the WSDL/Schema targetNamespace to package mapping as defined by the Jakarta XML Binding 2.1 spec.	`-p`
`target`	Generate code as per the given JAXWS specification version. For example, 2.0 value generates compliant code for JAXWS 2.0 spec.	No. Defaults to 2.2.	`-target`
`quiet`	Suppress **wsimport** output.	No. Defaults to false.	`-quiet`
`fork`	Used to execute wsgen using forked VM.	No. Defaults to false.	None.
`failonerror`	Indicates whether errors will fail the build.	No. Defaults to true.	None.

Attribute	Description	Required	Command line
`xadditionalHeaders`	Map headers not bound to request or response message to Java method parameters.	No. Defaults to false.	`-XadditionalHeaders`
`xauthfile`	File to carry authorization information in the format `http://username:password@example.org/stock?wsdl`. The asterisk character ("*") can be used to match multiple URL patterns.	No. Defaults to `$HOME/.metro/auth`.	`-Xauthfile`
`xdebug`	Print debug information.	No. Defaults to false.	`-Xdebug`
`xdisableAuthenticator`	Disables Authenticator used by Eclipse Implementation of XML Web Services, `-Xauthfile` option will be ignored if `-XdisableAuthenticator` is set.	No. Defaults to false.	`-XdisableAuthenticator`
`xNoAddressingDatabinding`	Enable binding of W3C EndpointReferenceType to Java.	No. Defaults to false.	`-Xno-addressing-databinding`
`xnocompile`	Do not compile generated Java files.	No. Defaults to false.	`-Xnocompile`

3.2.3. Nested Elements

wsimport supports the following nested element parameters.

3.2.3.1. `binding`

To specify more than one external binding file at the same time, use a nested `<binding>` element, which has the same syntax as `<fileset>` [http://ant.apache.org/manual/Types/fileset.html].

3.2.3.2. `arg`

Additional command line arguments passed to the **wsimport**. For details about the syntax, see the relevant section [http://ant.apache.org/manual/using.html#arg] in the Ant manual. This nested element can be used to specify various options not natively supported in the **wsimport** Ant task. For example, currently there is no native support for the following **wsimport** command-line options:

- `-XdisableSSLHostnameVerification`

This nested element can be used to pass the `-X` command-line options directly like "-XadditionalHeaders". To use any of these features from the **wsimport** Ant task, you must specify the appropriate nested `<arg>` elements.

3.2.3.3. `xjcarg`

The usage is similar to `<arg>` nested element, except that these arguments are directly passed to the XJC tool (Jakarta XML Binding Schema Compiler), which will be used for compiling the schema ref-

erenced in the wsdl. For details about the syntax, see the relevant section [http://ant.apache.org/manual/using.html#arg] in the Ant manual.

3.2.3.4. `javacarg`

The usage is similar to `<arg>` nested element, except that these arguments are directly passed to the Javac Compiler, which will be used for compiling sources generated during processing the referenced wsdl. For details about the syntax, see the relevant section [http://ant.apache.org/manual/using.html#arg] in the Ant manual.Note: use '=' instead of space as a delimiter between option name and its value.

3.2.3.5. `jvmarg`

Use nested `<jvmarg>` elements to specify arguments for the the forked VM (ignored if fork is disabled), see the relevant section [http://ant.apache.org/manual/using.html#arg] in the Ant manual.

3.2.3.6. xmlcatalog

The xmlcatalog [http://ant.apache.org/manual/Types/xmlcatalog.html] element is used to resolve entities when parsing schema documents.

3.2.3.7. depends/`produces`

Files specified with this nested element are taken into account when the task does a modification date check. For proper syntax, see `<fileset>` [http://ant.apache.org/manual/Types/fileset.html].

3.3. wsimport Examples

```
<wsimport
        destdir="${build.classes.home}"
        debug="true"
        wsdl="AddNumbers.wsdl"
        binding="custom.xml"/>
```

The above example generates client-side artifacts for `AddNumbers.wsdl`, stores `.class` files in the `${build.classes.home}` directory using the `custom.xml` customization file. The classpath used is `xyz.jar` and compiles with debug information on.

```
<wsimport
        keep="true"
        sourcedestdir="${source.dir}"
        destdir="${build.classes.home}"
        extension="true"
        wsdl="AddNumbers.wsdl">
    <xjcarg value="-cp"/>
    <xjcarg file="path/to/fluent-api.jar" />
    <xjcarg value="-Xfluent-api"/>
</wsimport>
```

The above example shows how to generates artifacts for `AddNumbers.wsdl` while passing options to Jakarta XML Binding **xjc** tool for XML Schema to Java compilation using your Jakarta XML Binding plugin. Note extension attribute which is set to `true`. You need to set this to use Jakarta XML Binding plugins.

Multiple Jakarta XML Web Services and Jakarta XML Binding binding files can be specified using `-b` option and they can be used to customize various things like package names, bean names, etc. More

information on Jakarta XML Web Services and Jakarta XML Binding binding files can be found in the WSDL Customization.

4. wsgen

4.1. wsgen Overview

The **wsgen** tool generates Jakarta XML Web Services portable artifacts used in Jakarta XML Web Services web services. The tool reads a web service endpoint class and generates all the required artifacts for web service deployment, and invocation. Eclipse Implementation of XML Web Services 2.3.3 also provides a **wsgen** Ant Task.

4.2. Launching wsgen

- **Solaris/Linux**
  - `JAXWS_HOME/bin/wsgen.sh -help`
- **Windows**
  - `JAXWS_HOME\bin\wsgen.bat -help`

4.3. wsgen Syntax

`wsgen` [OPTION]... <SEI>

The following table lists the **wsgen** options:

Option	Description
`-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.
`-J<javacOption>`	Pass this option to Javac compiler. Note: use '=' instead of space as a delimiter between option name and its value.
`-keep`	Keep generated files.
`-r <directory>`	Used only in conjunction with the `-wsdl` option. Specify where to place generated resource files such as WSDLs.
`-s <directory>`	Specify where to place generated source files.
`-encoding <encoding>`	Set the encoding name for generated sources, such as EUC-JP or UTF-8. If `-encoding` is not specified, the platform default encoding is used.
`-verbose`	Output messages about what the compiler is doing.

Option	Description
`-version`	Print version information. Use of this option will ONLY print version information. Normal processing will not occur.
`-fullversion`	Print full version information. Use of this option will ONLY print version information. Normal processing will not occur.
`-wsdl[:protocol]`	By default **wsgen** does not generate a WSDL file. This flag is optional and will cause **wsgen** to generate a WSDL file and is usually only used so that the developer can look at the WSDL before the endpoint is deploy. The `protocol` is optional and is used to specify what protocol should be used in the `wsdl:binding`. Valid protocols include: `soap1.1` and `Xsoap1.2`. The default is `soap1.1`. `Xsoap1.2` is not standard and can only be used in conjunction with the `-extension` option.
`-servicename <name>`	Used only in conjunction with the `-wsdl` option. Used to specify a particular `wsdl:service` name to be generated in the WSDL. Example: `-servicename "{http://mynamespace/}MyService"`
`-portname <name>`	Used only in conjunction with the `-wsdl` option. Used to specify a particular `wsdl:port` name to be generated in the WSDL. Example: `-portname "{http://mynamespace/}MyPort"`
`-inlineSchemas`	Used to inline schemas in a generated wsdl. Must be used in conjunction with the `-wsdl` option.
`-x <path>`	Used to specify External Web Service Metadata xml descriptor to be used. If there are more such files, the option must be used before each of those.
`-Xnocompile`	Do not compile generated Java files.

4.4. wsgen Example

```
wsgen -d stock -cp myclasspath stock.StockService
```

This will generate the wrapper classes needed for `StockService` annotated with `@WebService` annotation inside the `stock` directory.

```
wsgen -wsdl -d stock -cp myclasspath stock.StockService
```

This will generate a SOAP 1.1 WSDL and schema for your Java class `stock.StockService` annotated with `@WebService` annotation.

```
wsgen -wsdl:Xsoap1.2 -d stock -cp myclasspath stock.StockService
```

Will generate a SOAP 1.2 WSDL.

Note

You don't have to generate WSDL at the development time as Jakarta XML Web Services runtime will automatically generate a WSDL for you when you deploy your service.

5. wsgen Ant Task

5.1. wsgen Task Overview

`wsgen` generates Jakarta XML Web Services portable artifacts used in Jakarta XML Web Services web services. The tool reads a web service endpoint class and generates all the required artifacts for web service deployment, and invocation.

5.2. Using wsgen Task

Before this task can be used, a `<taskdef>` element needs to be added to the project as given below:

```
<taskdef name="wsgen" classname="com.sun.tools.ws.ant.WsGen">
    <classpath path="jaxws.classpath"/>
</taskdef>
```

where `jaxws.classpath` is a reference to a path-like structure [http://ant.apache.org/manual/using.html#path], defined elsewhere in the build environment, and contains the list of classes required by the Eclipse Implementation of XML Web Services tools.

5.2.1. Environment Variables

- ANT_OPTS [http://wiki.apache.org/ant/TheElementsOfAntStyle] - command-line arguments that should be passed to the JVM. For example, you can define system properties or set the maximum Java heap size here.

5.2.2. wsgen Task Attributes

The attributes and elements supported by the Ant task are listed below:

```
<wsgen
        sei="..."
        destdir="directory for generated class files"
        classpath="classpath" | cp="classpath"
        resourcedestdir="directory for generated resource files such as
 WSDLs"
        sourcedestdir="directory for generated source files"
        keep="true|false"
        encoding="..."
        verbose="true|false"
        genwsdl="true|false"
        protocol="soap1.1|Xsoap1.2"
        servicename="..."
        portname="..."
        extension="true|false"
        inlineSchemas="true|false"
        fork="true|false"
        failonerror="true|false"
        xnocompile="true|false">
    <classpath refid="..."/>
    <externalmetadata file="..."/>
```

```
    <javacarg value="..."/>
    <jvmarg value="..."/>
</wsgen>
```

Attribute	Description	Required	Command line
`sei`	Name of the service endpoint implementation class.	Yes.	SEI
`destdir`	Specify where to place output generated classes.	No. Defaults to current working directory.	`-d`
`classpath`	Specify where to find input class files.	One of these or nested `classpath` element.	`-classpath`
`cp`	Same as `-classpath`.		`-cp`
`resourcedestdir`	Used only in conjunction with the -wsdl option. Specify where to place generated resource files such as WSDLs.	No. Defaults to `destDir`.	`-r`
`sourcedestdir`	Specify where to place generated source files.	No. Defaults to current working directory.	`-s`
`encoding`	Set the encoding name for generated sources, such as UTF-8.	No. Defaults to platform default.	`-encoding`
`keep`	Keep generated files.	No. Defaults to false.	`-keep`
`verbose`	Output JAX_WS RI version and messages about what the compiler is doing.	No. Defaults to false.	`-verbose`
`genwsdl`	Specify that a WSDL file should be generated.	No. Defaults to false.	`-wsdl`
`protocol`	Used in conjunction with `genwsdl` to specify the protocol to use in the `wsdl:binding`. Value values are `"soap1.1"` or `"Xsoap1.2"`, default is `"soap1.1"`. `"Xsoap1.2"` is not standard and can only be used in conjunction with the -extensions option.	No. Defaults to `-wsdl:soap11`.	`-wsdl[:protocol]`
`servicename`	Used in conjunction with the genwsdl option. Used to specify a particular `wsdl:service` name for the generated WSDL. Example: `servicename="{http://mynamespace/}MyService"`	No. No defaults.	`-servicename`

Attribute	Description	Required	Command line
`portname`	Used in conjunction with the genwsdl option. Used to specify a particular `wsdl:portname` name for the generated WSDL. Example: `portname="{http://mynamespace/}MyPort"`	No. No defaults.	`-portname`
`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.	No. Defaults to false.	`-extension`
`inlineSchemas`	Used to inline schemas in a generated wsdl. Must be used in conjunction with the `wsdl` option.	No. Defaults to false.	`-inlineSchemas`
`fork`	Used to execute wsgen using forked VM.	No. Defaults to false.	None.
`failonerror`	Indicates whether errors will fail the build.	No. Defaults to true.	None.
`xnocompile`	Do not compile generated Java files.	No. Defaults to false.	`-Xnocompile`

The `classpath`/`cp` attribute is a path-like structure [http://ant.apache.org/manual/using.html#path] and can also be set via nested `<classpath>` elements.

5.2.3. Nested Elements

wsgen supports the following nested element parameters.

5.2.3.1. `external-metadata`

An optional element `external-metadata` can be used if there are any web service metadata to be specified in xml file instead of java annotations:

```
<wsgen
        resourcedestdir="${wsdl.dir}"
        sei="fromjava.server.AddNumbersImpl">
    <classpath refid="compile.classpath"/>
    <external-metadata file="metadata-AddNumbersImpl.xml"/>
</wsgen>
```

For details see External Web Service Metadata.

5.2.3.2. `javacarg`

The usage is similar to `<arg>` nested element, except that these arguments are directly passed to the Javac Compiler, which will be used for compiling sources generated during processing the referenced wsdl. For

details about the syntax, see the relevant section [http://ant.apache.org/manual/using.html#arg] in the Ant manual. Note: use '=' instead of space as a delimiter between option name and its value.

5.2.3.3. `jvmarg`

Use nested `<jvmarg>` elements to specify arguments for the the forked VM (ignored if fork is disabled), see the relevant section [http://ant.apache.org/manual/using.html#arg] in the Ant manual.

5.3. wsgen Task Examples

```
<wsgen
        resourcedestdir="${wsdl.dir}"
        sei="fromjava.server.AddNumbersImpl">
    <classpath refid="compile.classpath"/>
</wsgen>
```

6. Annotation Processing

6.1. javac annotation processing

As an improvement for the **apt** Pluggable Annotation Processing API [http://www.jcp.org/en/jsr/detail?id=269] was introduced in Java SE 6. It provides an API to allow the processing of JSR 175 [http://www.jcp.org/en/jsr/detail?id=175] annotations (metadata); this will require modeling elements of the *Java* TM *programming language* as well as processing-specific functionality.

The **javac** generates the portable artifacts used in Jakarta XML Web Services services.

6.2. javac Annotation Processing Syntax

`javac` [JAVAC_OPTION]... <SOURCE_FILE>...

The following table lists the **javac** options useful for annotation processing:

Option	Description
`-classpath <path>`	Specifies where to find user class files and annotation processor factories.
`-d <path>`	Specifies where to place processor generated class files.
`-s <path>`	Specifies where to place processor generated source files.
`-source <release>`	Provide source compatibility with the specified release.
`-Xprint`	Print out textual representation of specified types for debugging purposes; perform neither annotation processing nor compilation. The format of the output may change.
`-XprintProcessorInfo`	Print information about which annotations a processor is asked to process.
`-XprintRounds`	Print information about initial and subsequent annotation processing rounds.

Option	Description
`-A[key[=value]]`	Options to pass to annotation processors.
`-proc:{none,only}`	Control whether annotation processing and/or compilation is done.
`-sourcepath <path>`	Specify where to find input source files.
`-endorseddirs <dirs>`	Override location of endorsed standards path.
`-processor <class1>[,<class2>...]`	Names of the annotation processors to run; bypasses default discovery process
`-processorpath <path>`	Specify where to find annotation processors

More options and detailed information on **javac** can be found here [http://docs.oracle.com/javase/6/docs/technotes/tools/windows/javac.html].

It is important when using **javac** with Jakarta XML Web Services to specify all of the JAR files in the distributed Jakarta XML Web Services bundle in the classpath passed to **javac**. As Jakarta XML Web Services annotation processor you have to specify `com.sun.tools.ws.processor.modeler.annotation.WebServiceAp`. The `-sourcepath <path>` option must also be provided so that **javac** and the Jakarta XML Web Services annotation processor can find all types referenced by a web service endpoint implementation class. In case you don't have additional annotation processors it's recommended to use **annotationProcessing** Ant Task.

7. annotationProcessing Ant Task

7.1. annotationProcessing Task Overview

An Ant task for the **annotationProcessing** is provided with Eclipse Implementation of XML Web Services 2.3.3.

7.2. annotationProcessing Task Attributes

The attributes and elements supported by the Ant task almost the same as for the javac ant task. With only one restriction: by default jax-ws annotation processor will be used. There several additional attributes for usability improvement:

Attribute	Description	Required
`procOnly`	Control whether annotation processing and compilation is done. Can be `true` or `false`. Default is `false`.	`false`
`sourcedestdir`	Specify where to place processor generated source files. `-s <path>`	`true`

The list of source files to be processed are specified via a nested `<srcdir>` element. That is, a path-like structure [http://ant.apache.org/manual/using.html#path]. The classpath attribute is a path-like structure [http://ant.apache.org/manual/using.html#path] and can also be set via nested `<classpath>` elements. Before this task can be used, a `<taskdef>` element needs to be added to the project as given below:

```
<taskdef name="annotationProcessing"
```

```
        classname="com.sun.tools.ws.ant.AnnotationProcessingTask">
    <classpath refid="jaxws.classpath"/>
</taskdef>
```

where `jaxws.classpath` is a reference to a path-like structure [http://ant.apache.org/manual/using.html#path], defined elsewhere in the build environment, and contains the list of classes required by the Jakarta XML Web Services tools.

7.3. annotationProcessing Task Examples

```
<annotationProcessing
        destdir="${build.classes.home}"
        sourceDestDir="${build.classes.home}"
        srcdir="${basedir}/src"
        includes="*.java"
        sourcepath="${basedir}/src">
    <classpath refid="jaxws.classpath"/>
</annotationProcessing>
```

The above example processes the Java source files in the `${basedir}/src` directory and generates the source and class files in `${build.classes.home}`. `${basedir}/src` is directory used to search for source files for multiple **annotationProcessing** rounds. The classpath is a reference to a path-like structure [http://ant.apache.org/manual/using.html#path] `jaxws.classpath`, defined elsewhere in the build environment.

```
<annotationProcessing
        debug="true"
        verbose="true"
        destdir="${build.classes.home}"
        srcdir="${basedir}/src""
        includes="**/server/*.java"
        sourceDestDir="${build.classes.home}"
        sourcepath="${basedir}/src">
    <classpath refid="jaxws.classpath"/>
</annotationProcessing>
```

The above example processes the Java source files in `${basedir}/src/**/server`, generates the source and class files in `${build.classes.home}`, compiles with debug information on, prints a message about what the compiler is doing. `${basedir}/src` is the directory used to search for source files for multiple **annotationProcessing** rounds. The classpath is a reference to a path-like structure [http://ant.apache.org/manual/using.html#path] `jaxws.classpath`, defined elsewhere in the build environment. This will also fork off the **annotationProcessing** process using the default `java` executable.

7.4. Apt Ant task

For backwards compatibility **Apt** Ant task still exists, but is marked as deprecated, and can be used with certain restrictions.

8. apt

The **apt** tool provided a facility for programmatically processing the annotations added to Java by JSR 175 [http://www.jcp.org/en/jsr/detail?id=175], *Metadata Facility for the Java* TM *Programming Language*. In brief, JSR 175 allows programmers to declare new kinds of structured modifiers that can be associated with program elements, fields, methods, classes, etc.

The **javac** completely replaced functionality provided by **apt**. It was done within implementation of JSR 269 [http://www.jcp.org/en/jsr/detail?id=269], *Pluggable Annotation Processing API*. So it's expected programmers to use **javac** instead of **apt**.

Using **javac** with Jakarta XML Web Services annotation processor will generate portable artifacts used in Jakarta XML Web Services services.

Be aware that the **apt** tool and its associated API, **com.sun.mirror.***, are being deprecated in JDK 7.

For more information on this compiler please see **Annotation Processing**.

FAQ

Table of Contents

1. Does Jakarta XML Web Services 2.0 support JAX-RPC 1.X?

No. Although, Jakarta XML Web Services's roots come from JAX-RPC, Jakarta XML Web Services is a completely different component than JAX-RPC.

2. What is the difference between JAX-RPC and Jakarta XML Web Services ?

One of the main difference between JAX-RPC and Jakarta XML Web Services is the programming model. A Jakarta XML Web Services based service uses annotations (such @WebService) to declare webservice endpoints. Use of these annotations obviates the need for deployment descriptors. With Jakarta XML Web Services, you can have a webservice deployed on a Java EE compliant application server without a single deployment descriptor. Apart from these, other additional features (such asynchronous callbacks etc) are also present.

3. Can a Jakarta XML Web Services and a JAX-RPC based service co-exist?

Yes.

4. Is it downloadable from maven repository ?

Yes from https://maven.java.net/content/repositories/releases/com/sun/xml/ws.

5. How do I find out which version of the Eclipse Implementation of XML Web Services I'm using?

Run the following command

```
$ wsgen or wsimport -version
```

Alternatively, each Jakarta XML Web Services jar has version information in its META-INF/MANIFEST.MF.

6. How can I change the Web Service address dynamically for a request ?

```
((BindingProvider)proxy).getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
 "...");
```

7. How do I do basic authentication in Jakarta XML Web Services ?

You can do the following:

```
HelloService service = new HelloService();
Hello proxy = (service.getHelloPort());
((BindingProvider)proxy).getRequestContext().put(BindingProvider.USERNAME_PROPERTY,
 "userfoo");
((BindingProvider)proxy).getRequestContext().put(BindingProvider.PASSWORD_PROPERTY,
 "passbar");
```

USERNAME_PROPERTY, PASSWORD_PROPERTY are used primarily for service requests. I think when you instantiate Service, it fetches WSDL and the server is returning 401. You could try any one of the following solutions.

- Use java.net.Authenticator class in your client application.
- Provide a local access to the WSDL using catalog. There is a catalog sample in the jax-ws distribution.
- Configure web.xml to allow GET requests without authentication

8. Which standards are supported by Eclipse Implementation of XML Web Services?

- Web Services Addressing 1.0 - Core
- Web Services Addressing 1.0 - SOAP Binding
- Web Services Addressing 1.0 - Metadata
- Web Services Addressing 1.0 - WSDL Binding (RI specific support)
- WS-Addressing - Member Submission
- SOAP 1.1 and 1.2
- REST and XML/HTTP

- WS-I Basic Profile 1.2 and 2.0
- WS-I Simple SOAP Binding Profile 1.0
- WS-I Attachment Profile 1.0
- MTOM

Extensions

Table of Contents

This page contains information about Eclipse Implementation of XML Web Services 2.3.3 specific features and extensions:

1. Sending and Receiving SOAP Headers

At times you need a way to send and receive SOAP headers in your message - these headers may not be defined in the WSDL binding but your application needs to do it anyway. One approach has been to write a `SOAPHandler` to do it, but its more work and is expensive as `SOAPHandlers` work on `SOAPMesssage` which is DOM based and Eclipse Implementation of XML Web Services runtime would need to do conversion from its abstract `Message` representation to `SOAPMessage` and vice versa.

There is a way to do it on the client side by downcasting the proxy to `WSBindingProvider` and use methods on it.

1.1. Sending SOAP Headers

You would downcasting the proxy to `WSBindingProvider` and set the `Outbound` headers.

```
HelloService helloService = new HelloService();
HelloPort port = helloService.getHelloPort();
WSBindingProvider bp = (WSBindingProvider) port;

bp.setOutboundHeaders(
        // simple string value as a header, like stringValue
        Headers.create(new QName("simpleHeader"), "stringValue"),
        // create a header from Jakarta XML Binding object
        Headers.create(jaxbContext, myJaxbObject));
```

1.2. Receiving SOAP Headers

```
List<Header> inboundHeaders = bp.getInboundHeaders();
```

2. Message logging

Web Services developers generally need to see SOAP Messages that are transferred between client and service for debugging. There are SOAP Monitors for this job, but you need modify the client or server code to use those tools. Eclipse Implementation of XML Web Services provides logging of SOAP messages

2.1. On the client

Set system property

```
com.sun.xml.ws.transport.http.client.HttpTransportPipe.dump=true
```

2.2. On the server side

Set system property

```
com.sun.xml.ws.transport.http.HttpAdapter.dump=true
```

3. Propagation of Server-side Stacktrace

This is a very useful feature while developing Web Services. Often the soap fault messages for not user defined faults does not convey enough information about the problem. Eclipse Implementation of XML Web Services relieves you from digging out the server logs to find out the stacktrace. Whole stacktrace (including nested exceptions) can be propagated in the SOAP Fault and the complete exception stacktrace can be made visible to the client as cause of `SOAPFaultException`.

3.1. Enabling propagation of Server-side stacktrace

Propagation of Stack trace is off by default. To turn it on for your Web Service Application to send the complete stack trace, set the system property

```
com.sun.xml.ws.fault.SOAPFaultBuilder.captureStackTrace=true
```

Samples

Table of Contents

This document explains the samples that are bundled along with Eclipse Implementation of XML Web Services 2.3.3.

The samples included with Eclipse Implementation of XML Web Services that is distributed on GitHub documents how to use Eclipse Implementation of XML Web Services in a non-Jakarta EE servlet container using a proprietary deployment descriptor `sun-jaxws.xml` and servlet `com.sun.xml.ws.transport.http.servlet.WSServlet`. This means that you can these Eclipse Implementation of XML Web Services applications in any servlet container that has been enabled with Eclipse Implementation of XML Web Services. Applications that use the proprietary DD and servlet will run in a Eclipse Implementation of XML Web Services enabled Jakarta EE servlet container, but they will be non-portable. If you wish to use these Eclipse Implementation of XML Web Services samples in a Jakarta EE container in a Jakarta EE portable manner you need to modify them to use the standard Jakarta EE deployment descriptor; please refer to the Jakarta EE [https://jakarta.ee/specifications/platform/] or GlassFish [https://eclipse-ee4j.github.io/glassfish/] documentation/samples.

All these samples are tested to run on GlassFish v5.x and on Apache Tomcat 5.x.

1. Directory Structure

This section explains the directory structure of the `samples` directory in the bundle:

Table 29. List of Samples

Sample	Description
`docs`	jax-ws-ri-overview, this file
`asyncservice`	Demonstrates Eclipse Implementation of XML Web Services specific endpoint to achieve server side asynchrony. This sample requires Servlet 3.0 API supported container like GlassFish 5.x.
`wsimport_clientjar`	Demonstrates use of `-clientjar` option of **wsimport**.
`optional_webxml`	Demonstrates the simplified deployment of web services using Eclipse Implementation of XML Web Services deployment by not requiring the configuration of web.xml.
`fromwsdl-wsaddressing-policy`	Starting from WSDL shows how to develop a WS-Addressing enabled endpoint using standardized Addressing Metadata support.
`fromjava-wsaddressing`	Starting from Java endpoint how to develop W3C WS-Addressing endpoint.

Sample	Description
`fromwsdl-wsaddressing`	Starting from WSDL shows how to develop a WS-Addressing endpoint using WS-Addressing WSDL Binding. This sample uses non-standard wsdl extension, also see standards based fromwsdl-wsaddressing-policy sample.
`fromjava`	Demonstrates how to build, deploy, and invoke a simple Web service starting from a Java service endpoint implementation using annotations.
`fromjavahandler`	Same as fromjava sample but with a simple logging handler on the client and server.
`fromwsdl`	Demonstrates how to build, deploy, and invoke a simple Web service starting from a WSDL using external customizations.
`fromwsdl_secure`	Same as fromwsdl sample but demonstrates how to build, deploy, and invoke an HTTPS Web service and client from a WSDL.
`fromwsdlhandler`	Same as fromwsdl sample but with a simple logging handler on the client and server.
`dispatch`	Demonstrates how to dynamically invoke web service endpoints.
`provider`	Demonstrates how to build, deploy, and invoke `javax.xml.ws.Provider` based Web service endpoint.
`asyncprovider`	Demonstrates how to build, deploy, and invoke a server side asynchronous `Provider` based Web service endpoint.
`annotations`	Same as fromjava but shows how to specify a different parameter name, operation name, targetNamespace, and other similar features.
`async`	Demonstrates how to generate async operations in a client SEI from a WSDL and invoke it from the client application.
`efficient-handler`	Demonstrates efficient handler using `com.sun.xml.ws.api.handler.MessageHandler`, which is an efficient protocol handler.
`external-customize`	Demonstrates how a client client application can customize a published WSDL using external binding file.
`external-metadata-fromjava`	Demonstrates how to develop web service without using java annotations - metadata are being provided with xml descriptors. This allows to expose as a web service already deployed implementations. This sample uses fromjava (better to say from java binary) approach.
`external-metadata-fromwsdl`	Demonstrates how to develop web service without using java annotations - metadata are being provided with xml descriptors. This allows to expose as

Sample	Description
	a web service already deployed implementations. This sample uses fromwsdl approach.
`inline-customize`	Demonstrates how a client application and server endpoint can be generated from a WSDL with embedded binding declarations.
`mtom`	Demonstrates how to enable MTOM and swaRef.
`mtom-large`	Demonstrates how to transfer large attachments in streaming fashion using MTOM.
`mtom-soap12`	Same as mtom sample but shows how to specify SOAP 1.2 binding.
`fromjava-soap12`	Same as fromjava sample but shows how to specify SOAP 1.2 binding.
`fromwsdl-soap12`	Same as fromwsdl sample but shows how to specify SOAP 1.2 binding.
`supplychain`	Same as fromjava sample but using JavaBeans as parameter and return types. Also the service can be built and deployed using Endpoint API.
`mime`	Demonstrates how a MIME binding is defined in the WSDL to send `wsdl:part` as MIME attachments. This requires that the development model is 'starting from WSDL'.
`mime-large`	Demonstrates how to transfer large attachment in streaming fashion using `wsi:swaref`.
`wsimport_catalog`	Demonstrates a how a WSDL and schema URI's can be resolved using catalog mechanism using **wsimport** ant tasks' catalog attribute and also using ant's core type `xmlcatalog`.
`catalog`	Shows the catalog capability on the client side; Catalog is used every time the implementation tries to access a resource identified by URI that is believed to contain either a WSDL document or any other document on which it depends .
`restful`	Shows an example of a REST Web Service implemented as a Jakarta XML Web Services `Provider` and accessed via a Jakarta XML Web Services `Dispatch` client. The `Request` uses an HTTP `GET` Request Method and uses the Jakarta XML Web Services `MessageContext` properties `PATH_INFO` and `QUERY_STRING`.
`stateful`	This sample shows the Eclipse Implementation of XML Web Services's stateful webservice support feature.
`schema_validation`	This sample demonstrates Eclipse Implementation of XML Web Services's validation feature that validates the incoming/outgoing messages are as per schema.

Sample	Description
`dual-binding`	This sample demonstrates the Eclipse Implementation of XML Web Services's featuer to expose the same service class under `SOAP/HTTP` and `XML/HTTP` binding.
`large_upload`	This sample demonstrates uploading a large file to the server.
`type_substitution`	This sample demonstrates type substitution and sending java types that are not directly referenced by the WSDL.
`xmlbind_datasource`	This sample demonstrates a REST based web-services using `XML/HTTP` binding along with `Provider`/`Dispatch`.

2. Prerequisites

Here is the list of prerequisites that needs to be met before any of the samples can be invoked:

1. Download Java SE 8 or later. Set `JAVA_HOME` to the Java SE installation directory.
2. Set `JAXWS_HOME` to the Eclipse Implementation of XML Web Services installation directory.

3. Installing Eclipse Implementation of XML Web Services 2.3.3

- Follow the Installation Instructions from Release Notes.
- Make sure that your container is configured for port 8080 as samples are hardcoded with this port info. Otherwise, you need to change samples to use the correct port by replacing '8080" with your port in all the files included in the sample.

4. Running the sample

Each sample has a `readme.txt` in its directory that details the instructions specific to the sample. Each sample can be built, deployed and invoked using the `ANT_HOME/bin/ant` and `build.xml` ant script in the root directory of the sample. Each ant script supports the following set of targets:

Target	Description
`server`	Builds and deploy the service endpoint WAR on GlassFish v5.x installation referenced by `$AS_HOME`
`server -Dtomcat=true`	Builds and deploy the service endpoint WAR on Tomcat installation referenced by `$CATALINA_HOME`
`client`	Builds the client
`run`	Runs the client

Some samples (e.g. fromjava, supplychain) can be built, deployed using `javax.xml.ws.Endpoint` API. These samples support extra targets:

Target	Description
`server-j2se`	Builds and deploys the `Endpoint` API based service endpoint (doesn't terminate until **server-j2se-stop** is called)
`server-j2se-stop`	Stops the `Endpoint` API based service endpoint (need to run from a different window)

It is essential for the service endpoint to be deployed on Application Server before clients can be built because clients use the WSDL exposed from the service endpoint deployed in the Application Server. So please make sure that your Application Server is either running before the `server` target is invoked or run it after the `server` target is invoked. You will have to wait a few minutes for the Application Server to deploy the service endpoint correctly before building the client.

We appreciate your feedback, please send it to metro-dev@eclipse.org [https://accounts.eclipse.org/mailing-list/metro-dev].